

STATE OF COLORADO

Intelligent Transportation Systems
700 Kipling Street, Suite 2500
Lakewood, Colorado 80215
Phone (303) 512-5834
FAX (303) 239-0848



INTEGRATION **Contract Routing No. 04 HAA 00063**

Communication **Server Detailed Design**

Version 1.0

Approved By

Frank Kinder
CDOT ITS Office

Signature: _____

Date: _____

John Williams
CDOT ITS Office

Signature: _____

Date: _____

Prepared By:



CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

Revision History

Date	Version	Description	Author
April 20, 2005	1.0	Initial Version	Ramesh Vellanki
March 12, 2005	1.1	Added extensions sections	Jason Westra

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

Table of Contents

i.	Executive Summary	1
ii.	Summary of Key Technologies	1
1.	Introduction	2
1.1	Purpose	2
1.2	Scope	2
1.3	Audience	2
1.4	Reference	2
2.	Key Design Concepts	2
2.1	Extensible	3
2.2	Asynchronous Communications	3
2.3	Clustering	3
2.4	Standards	3
2.5	Key Technologies	4
3.	Detailed Design	4
3.1	Comm API	4
3.1.1	Common Classes	5
3.1.1.1	Common Classes – Class Diagram	5
3.1.1.2	Cdot.ctms.layer.services.comm.CommException	6
3.1.1.3	Cdot.ctms.layer.services.comm.CommErrorException	6
3.1.1.4	Cdot.ctms.layer.services.comm.CommEvent	6
3.1.2	Common Client Interface	7
3.1.2.1	Common Client Interface – Class Diagram	7
3.1.2.2	Cdot.ctms.layer.services.comm.cci.InteractionContext	8
3.1.2.3	Cdot.ctms.layer.services.comm.cci.Interaction	9
3.1.2.4	Cdot.ctms.layer.services.comm.cci.InteractionFactory	9
3.1.2.5	Cdot.ctms.layer.services.comm.cci.Record	9
3.1.2.6	Cdot.ctms.layer.services.comm.cci.RecordFactory	9
3.1.2.7	cdot.ctms.layer.services.comm.cci.UnhandledInteractionException	10
3.1.3	Service Provider Interface	10
3.1.3.1	Service Provider Interface – Class Diagram	10
3.1.3.2	cdot.ctms.layer.services.comm.spi.ConfigurationException	11
3.1.3.3	cdot.ctms.layer.services.comm.spi.ConnectInfo	11
3.1.3.4	cdot.ctms.layer.services.comm.spi.Connection	12
3.1.3.5	cdot.ctms.layer.services.comm.spi.ConnectionEvent	12
3.1.3.6	cdot.ctms.layer.services.comm.spi.ConnectionEventListener	13
3.1.3.7	cdot.ctms.layer.services.comm.spi.ConnectionFactory	13
3.1.3.8	cdot.ctms.layer.services.comm.spi.PooledConnectInfo	13
3.2	CTMS Container	13
3.2.1	Notifications	14
3.2.2	Connection Pooling	14
3.2.3	Locking	15
3.2.4	Management	15
3.2.5	Container Execution Environment	15
3.2.6	Container	16
3.2.6.1	Container Class Diagrams	16

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

3.2.6.2	Container Interfaces	16
3.2.6.3	Container Implementation	17
3.2.6.4	Cdot.ctms.layer.services.comm.container.Container	17
3.2.6.5	Cdot.ctms.layer.services.comm.container.CTMSContainer	19
3.2.6.6	Cdot.ctms.layer.services.comm.container.ContainerManager	19
3.2.6.7	Cdot.ctms.layer.services.comm.container.CTMSContainerManager	20
3.2.7	Connection Management	20
3.2.7.1	Managed Connection Factory Class Diagram	20
3.2.7.2	Managed Connection Factory Sequence Diagrams	21
3.2.7.3	Cdot.ctms.layer.services.comm.ManagedConnection	24
3.2.7.4	Cdot.ctms.layer.services.comm.CTMSManagedConnection	24
3.2.7.5	Cdot.ctms.layer.services.comm.ManagedConnectionFactory	24
3.2.7.6	Cdot.ctms.layer.services.comm.CTMSManagedConnectionFactory	25
3.2.8	Connection Spec Classes	25
3.2.8.1	Cdot.ctms.layer.services.comm.connection.CTMSConnectionSpec	25
3.2.8.2	Cdot.ctms.layer.services.comm.connection.CTMSSerialConnectionSpec	26
3.2.8.3	Cdot.ctms.layer.services.comm.connection.CTMSModemConnectionSpec	26
3.2.8.4	Cdot.ctms.layer.services.comm.connection.CTMSSocketConnectionSpec	26
3.2.9	Connection Classes	27
3.2.9.1	Cdot.ctms.layer.services.comm.connection.CTMSConnection	27
3.2.9.2	Cdot.ctms.layer.services.comm.connection.CTMSSerialConnection	28
3.2.9.3	Cdot.ctms.layer.services.comm.connection.CTMSModemConnection	28
3.2.9.4	Cdot.ctms.layer.services.comm.connection.CTMSSocketConnection	28
3.2.10	Lock Manager	28
3.2.10.1	Lock Manager Class Diagram	29
3.2.10.2	Cdot.ctms.layer.services.comm.container.LockManager	29
3.2.10.3	Cdot.ctms.layer.services.comm.container.CTMSLockManager	29
3.2.10.4	Cdot.ctms.layer.services.comm.container.Lockable	30
3.2.10.5	Cdot.ctms.layer.services.comm.container.Lock	30
3.2.10.6	Lock Manager - Lock Sequence Diagram	31
3.2.10.7	Lock Manager - Unlock Sequence Diagram	32
3.3	Protocol Adaptors	34
3.3.1	ProtocolAdaptor	34
3.3.1.1	HDLCProtocolAdaptor	34
3.3.1.2	PMPPProtocolAdaptor	34
3.3.1.3	Protocol Adapter Class Diagram	35
3.3.1.4	Protocol Adapter Encode Sequence Diagram	36
3.3.1.5	Protocol Adaptor Decode Sequence Diagram	37
3.3.1.6	Protocol Packet Diagram	38
3.3.2	Records	38
3.3.2.2	Record Class Diagram	39
3.3.3	SNMP	39
3.3.3.1	Purpose	39
3.3.3.2	SNMP Class Diagram	41
3.3.3.3	SNMP Encode Sequence Diagram	43
3.3.3.4	SNMP Decode Sequence Diagram	44
3.3.4	Management Information Database	44
3.3.4.1	Purpose	44
3.3.4.2	MIB Class Diagram	45
3.3.4.3	MIB Sequence Diagram	47
3.3.4.4	XML File Definition	47
3.4	Communication Services	48
3.4.1	Delegate channel classes	48

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

3.4.1.1	delegate.channel.CommChannelMDB	49
3.4.1.2	delegate.channel.comm.CTMSCommChannelContext	49
3.4.1.3	delegate.channel.comm.CTMSCommChannelFailedContext	49
3.4.1.4	delegate.channel.comm.CTMSCommChannelCmdFactory	49
3.4.1.5	delegate.channel.comm.CommSuccessCmd	49
3.4.1.6	delegate.channel.comm.CommFailedCmd	49
3.4.1.7	delegate.channel.comm.CTMSCommFailedDTO	50
3.4.1.8	Comm Channel Class Diagram	50
3.4.1.9	delegate.channel.comm.cmd Package	50
3.4.1.10	Comm Channel Cmd Class Diagram	51
3.4.1.11	Comm Success Event Sequence Diagram	52
3.4.1.12	Comm Failure Event Sequence Diagram	52
3.4.2	EJBs	53
3.4.2.1	Comm Data Facade Class Diagram	54
3.4.2.2	Cdot.ctms.layer.services.comm.facade.CommunicationSLSBean	54
3.4.2.3	Cdot.ctms.layer.services.comm.facade.DeviceServiceSLSBean	54
3.4.2.4	Cdot.ctms.layer.services.comm.facade.DmsServiceSLSBean	54
3.4.2.5	Cdot.ctms.layer.services.comm.data.device package	55
3.4.2.6	Cdot.ctms.layer.services.comm.data.dms package	55
3.4.2.7	Cdot.ctms.layer.services.comm.data.fieldcomm package	55
3.4.3	Composite DTOs	55
3.4.3.1	Cdot.ctms.layer.services.comm.data.DeviceModelCDTO	55
3.4.3.2	Cdot.ctms.layer.services.comm.data.DmsPollModelCDTO	55
3.4.4	Delegate Actions	55
3.4.4.1	Delegate Actions Class Diagram	56
3.4.4.2	Cdot.ctms.layer.delegate.actions.CommAction	56
3.4.4.3	Cdot.ctms.layer.delegate.actions.DmsAction	57
3.4.4.4	Cdot.ctms.layer.delegate.actions.LibraryAction	57
3.4.4.5	Cdot.ctms.layer.delegate.actions.DeviceAction	57
3.5	Extending the Comm Server	57
3.5.1	Adding Vendor-specific Extensions	57
3.5.2	cdot.ctms.layer.services.comm.cci.InteractionFactory	57
3.5.2.1	Interaction	58
3.5.2.2	InteractionFactory	58
3.5.2.3	Connection	58
3.5.2.4	ConnectionFactory	58
3.5.2.5	Create a Container Deployment Descriptor	59
3.5.3	Adding Custom Implementations	59
3.5.4	Adjusting Global and Container Settings	61
3.5.4.1	Global Properties	62
3.5.4.2	Local Properties	62
3.5.5	Add Custom Comm Server Event Listeners	63
3.5.5.1	Interaction Events	64

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

Table of Figures

Figure 1: Comm API - Common Class Diagram.....	6
Figure 2: Comm API – Common Client Interface Class Diagram	8
Figure 3: Comm API - Service Provider Interface Class Diagram.....	11
Figure 4: Container Services.....	14
Figure 5: Container Execution Environment	15
Figure 6: Container Interfaces Class Diagram.....	16
Figure 7: Container Implementation Class Diagram	17
Figure 8: CTMSManagedConnection Class Diagram	21
Figure 9: Get Connection 1 - Sequence Diagram	22
Figure 10: Get Connection 2 - Sequence Diagram	23
Figure 11: CTMS ConnectInfo Class Diagram	26
Figure 12: CTMS Connection Class Diagram.....	27
Figure 13: CTMSLockManager Class Diagram	29
Figure 14: Lock - Sequence Diagram	32
Figure 15: Unlock - Sequence Diagram.....	33
Figure 16: Protocol Adapter Class Diagram.....	35
Figure 17 Protocol Adapter Encode Sequence Diagram	36
Figure 18 Protocol Adaptor Decode Sequence Diagram.....	37
Figure 19 Protocol Packet Diagram.....	38
Figure 20 Record Class Diagram.....	39
Figure 21 SNMP Class Diagram.....	41
Figure 22 SNMP Encode Sequence Diagram.....	43
Figure 23 SNMP Decode Sequence Diagram.....	44
Figure 24 MIB Class Diagram.....	45
Figure 25 MIB Sequence Diagram	47
Figure 26 Comm Channel Class Diagram	50
Figure 27 Comm Channel Cmd Class Diagram	51
Figure 28 Comm Success Event Sequence Diagram.....	52
Figure 29 Comm Failure Event Sequence Diagram	53
Figure 30: Comm Data Façade Class Diagram.....	54
Figure 31: Delegate Actions Class Diagram.....	56
Figure 32: Add Connection Pool Sequence Diagram	56
Figure 32: Container Interfaces Table	60
Figure 33: Example Communications-jmx-service.xml	61
Figure 34: Example CTMSContainer.xml.....	61
Figure 35: Global Comm Server Properties Table	62
Figure 36: Local Comm Server Properties Table	63

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

Communications Server Detailed Design

i. Executive Summary

This document describes the detailed design of the Communications Server (Comm Server). It covers the core Comm API, default implementations, Container, processing flow of a successful and failed communication requests. It discusses how to extend the Comm Server for vendor-specific features and how to change the behavior of the server declaratively to customize the Comm Server for different needs.

ii. Summary of Key Technologies

The development Communications server core framework will employ proven technologies, software methodologies, and best practices. Specifically, the Rational Unified Process will be followed in order to facilitate a thorough analysis and design of the complex system. The core technology will be built using Java, EJB (Enterprise JavaBeans), J2CA (Java Connector Architecture), XML, Object-Oriented concepts, and several design patterns including Factory, Delegate, Model-View-Controller, Service Activator, and Service Locator.

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

1. Introduction

This document describes the detailed design of the Communications Server (Comm Server). It covers the core Comm API, default implementations, Container, processing flow of a successful and failed communication requests. It discusses how to extend the Comm Server for vendor-specific features and how to change the behavior of the server declaratively to customize the Comm Server for different needs.

1.1 Purpose

- ❑ To supplement the technical information provided in the Server Detailed Design document.
- ❑ The reader should understand the high-level classes and patterns that make up the Communications Server after reading this document.

1.2 Scope

The scope of this document is the CTMS/CTIS core Communications Server. It does not contain vendor-specific extensions. For vendor extensions, see the vendor-specific design documents.

1.3 Audience

- ❑ Architects
- ❑ Developers

1.4 Reference

The following references were either used in the creation of this document or may be used to supplement the detailed descriptions of the Communications Server.

- ❑ CTMS_Server_Detailed_Design.doc
- ❑ CTMS Database Design document
- ❑ Sun Microsystems' Java Management Extensions
- ❑ Sun Microsystems' Java 2 Enterprise Edition

2. Key Design Concepts

The following section describes the factors that the Comm Server designs take into account.

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

2.1 Extensible

Formatted: Bullets and Numbering

There are hundreds of devices that can be integrated into a single communications platform like the Comm Server; however, if the platform is not extensible, it will not be able to support new devices easily. The Comm Server uses a Container-based approach similar to component-models such as J2EE to provide plug-n-play capabilities. This dramatically decreases the time to integrate new products on top of the Comm Server's communications platform.

2.2 Asynchronous Communications

Formatted: Bullets and Numbering

The Comm Server performs all communications asynchronously using JMS as the message bus backbone. This is important since communications to devices are both time-consuming and inherently unstable. Take a device that is connected to a modem for example. To connect to the device, the Comm Server must dial-up the device's phone number over a modem and then communicate with it over a landline or wireless network, if the device uses a cell phone. Dial-up over a modem can take 30-60 seconds. This is too long for a caller to wait, so the Comm Server uses JMS to put the instruction on a queue and process it when it has bandwidth to do so. As for stability, static can make communications over landlines poor resulting in incorrect data sent or received. Wireless devices can drop calls as easily as cell phones do for us. So, using an asynchronous design allows for retrying device communications without worrying about holding up the caller in real-time.

2.3 Clustering

Clustering offers both high-availability as well as load balancing to prevent a single instance of an application from being inundated with requests. The Comm Server supports clustering when deployed within a clustered J2EE server. This design ensures communications are not solely dependent on a single instance of the Comm Server. Also, it supports clustered JMS as well as clustered communications on devices through distributed locking mechanisms. Together, these features provide a backbone that will allow the Comm Server to scale as far as the infrastructure will allow. For instance, if the Comm Server can only support a pool of modem connections as large as the modem bank. Any more and the Comm Server would wait anyway until a modem had completed its previous request anyway.

2.4 Standards

Formatted: Bullets and Numbering

The adherence to standards is what allows the Comm Server to be so flexible in supporting new devices as well as customizations. The following standards were adhered to in the design and development of the Communications Server:

- ❑ *SNMP* – The *Simple Network Management Protocol (SNMP)* is an application layer protocol that facilitates the exchange of management information between network devices. It is part of the Transmission Control Protocol/Internet Protocol (TCP/IP) protocol suite.
- ❑ *NTCIP* – The NTCIP is a family of standards that provides both the rules for communicating (called protocols) and the vocabulary (called objects) necessary to allow

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

electronic traffic control equipment from different manufacturers to operate with each other as a system. The NTCIP is the first set of standards for the transportation industry that allows traffic control systems to be built using a "mix and match" approach with equipment from different manufacturers. Therefore, NTCIP standards reduce the need for reliance on specific equipment vendors and customized one-of-a-kind software. To assure both manufacturer and user community support, NTCIP is a joint product of the National Electronics Manufacturers Association (NEMA), the American Association of State Highway and Transportation Officials (AASHTO), and the Institute of Transportation Engineers (ITE). The NTCIP originated as the National Transportation Communications for Intelligent Transportation System (ITS) Protocol (NTCIP). The NTCIP is part of a larger effort to develop a family of ITS standards.

2.5 Key Technologies

Formatted: Bullets and Numbering

Several technologies were leveraged to reduce system design and development time.

These tools include:

- *JoeSNMP-0.2.5* – JoeSNMP is a completely free implementation of the SNMP protocol written entirely in Java and is distributed as part of OpenNMS. JoeSNMP supports SNMP protocol versions 1 and 2c.
- *RXTX-2.1* – RXTX is a set of Java wrappers around native libraries providing serial and parallel communication for the Java Development Toolkit (JDK). RXTX is distributed under the GNU LGPL license.
- J2EE – The Java 2 Platform, Enterprise Edition (J2EE) defines the standard for developing multi-tier enterprise applications. The J2EE platform simplifies enterprise applications by basing them on standardized, modular components, by providing a complete set of services to those components, and by handling many details of application behavior automatically, without complex programming. The Comm Server uses JMS for messaging, JMX for management and plug-n-play deployment, and has based its Container on J2CA, the J2EE Connector Architecture.

3. Detailed Design

The following section describes the Comm Server in detail. It describes where the Comm Server fits into the overall CTMS architecture as well as a background on the Comm API, default implementations, and the Container's inner workings.

3.1 Comm API

The Comm API is a set of classes and interfaces that represent the base Comm Server. The Comm API was based on the J2CA API. While the problem J2CA (e.g., providing configurable "connectors" to external resources) was designed to solve was similar to that of the Comm Server, the implementation was not a good fit. J2CA is designed to connect one external resource such as a database per "connector". In other words, a connector defined how to connect to one and only one resource. On the other hand, the CTMS Comm Server needed to configure

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

multiple external resources (e.g., devices) per connector. For example, the same connection (e.g., a modem port in a modem bank) can be used to connect to multiple devices. In the case of modem-based devices, it is only the phone number that is different between these devices. Thus, there was an invalid relationship to overcome with directly applying J2CA to the Comm Server.

It was a goal of CTMS that anyone familiar with J2EE and more importantly, J2CA, would be comfortable with the design and development of Comm Server components. The Comm API is similar to JDBC and J2CA. It is customized for CTMS n-1 requirements and stripped down to remove unnecessary complexity. CTMS applied concepts from the two APIs like records, record factories, interactions, connections and connection factories. Also, like J2EE, CTMS uses JNDI for binding and looking up references to deployed Comm Server resources.

With that said, let's examine the breakdown of the classes and interfaces we call the Comm API. The Comm API includes common classes such as exceptions and utilities, CCI (common client interface) and SPI (service provider interface).

The Comm API spans the following packages:

1. `cdot.ctms.layer.services.comm`
2. `cdot.ctms.layer.services.comm.cci`
3. `cdot.ctms.layer.services.comm.spi`

The Comm Server has many default implementations of the Comm API. However, there are some that must be implemented when integrating a new device or type of connection. To help clarify interfaces that the Service Provider must implement, each interface below will have the following at the bottom of its description:

Implemented By: [CTMS or Service Provider]

This will describe whether or not it has a default CTMS implementation or whether the SP is required to provide an implementation for this interface in order for it to plug into the Comm Server.

3.1.1 Common Classes

Classes defined in the `cdot.ctms.layer.services.comm` package are considered the Comm API's common classes. They include `CommException`, `CommErrorException`, and `CommEvent`.

3.1.1.1 Common Classes – Class Diagram

The following diagram shows the classes and methods in the `cdot.ctms.layer.services.comm` package.

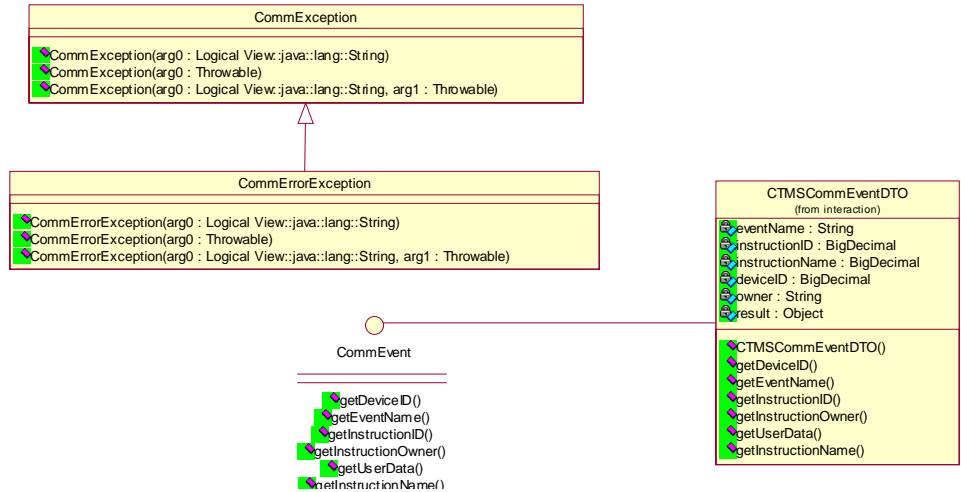


Figure 1: Comm API - Common Class Diagram

3.1.1.2 Cdot.ctms.layer.services.comm.CommException

This is a basic exception that represents a failure in the communications process. It can be thrown directly or specialized through sub-classing. It is described in the throws clause of nearly every Comm API interface. Vendors may want to subclass CommException and raise very specific information about a failure to the Comm Server.

3.1.1.3 Cdot.ctms.layer.services.comm.CommErrorException

This exception is a specialized CommException that represents a communication error has taken place during the connecting of a device or while sending and receiving data over a valid connection.

3.1.1.4 Cdot.ctms.layer.services.comm.CommEvent

CommEvent is an interface. Its implementation represents an event posted from the Comm Server. It holds basic information about the request that the Comm Server is or was processing and has support for custom, user-specific data. CommEvent is used to reply asynchronously to callers that have issued an instruction and are waiting for a reply back from the Comm Server with the results.

As shown in the class diagram above, CTMS provides a default implementation of CommEvent called `cdot.ctms.layer.services.comm.interaction.CTMSCommEventDTO`. It also implements the DTO pattern to allow it to be passed remotely to event listeners. Likewise, other implementations should follow RMI's rules for guaranteeing data serialization.

Implemented By: CTMS

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

3.1.2 Common Client Interface

Classes in the `cdot.ctms.layer.services.comm.cci` package represent the Common Client Interface to a vendor's devices. They provide a way to flexibly implement the logic to interact with devices and their data. CCI classes include: `Interaction`, `InteractionFactory`, `InteractionContext`, `Record`, and `RecordFactory`. The programming model is simple. It uses two well-known patterns, the Command and Factory patterns for extensibility.

Each time a new device operation is supported; a new `Interaction` is created. If the new `Interaction` requires new data definitions, a new `Record` will be created as well to represent the data sent and received during the course of the interaction. The `InteractionFactory` implementation is updated to support the creation of the new `Interaction` and if a new `Record` was created, the `RecordFactory` is updated as well.

3.1.2.1 Common Client Interface – Class Diagram

The following diagram shows the classes and methods in the `cdot.ctms.layer.services.comm.cci` package.

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

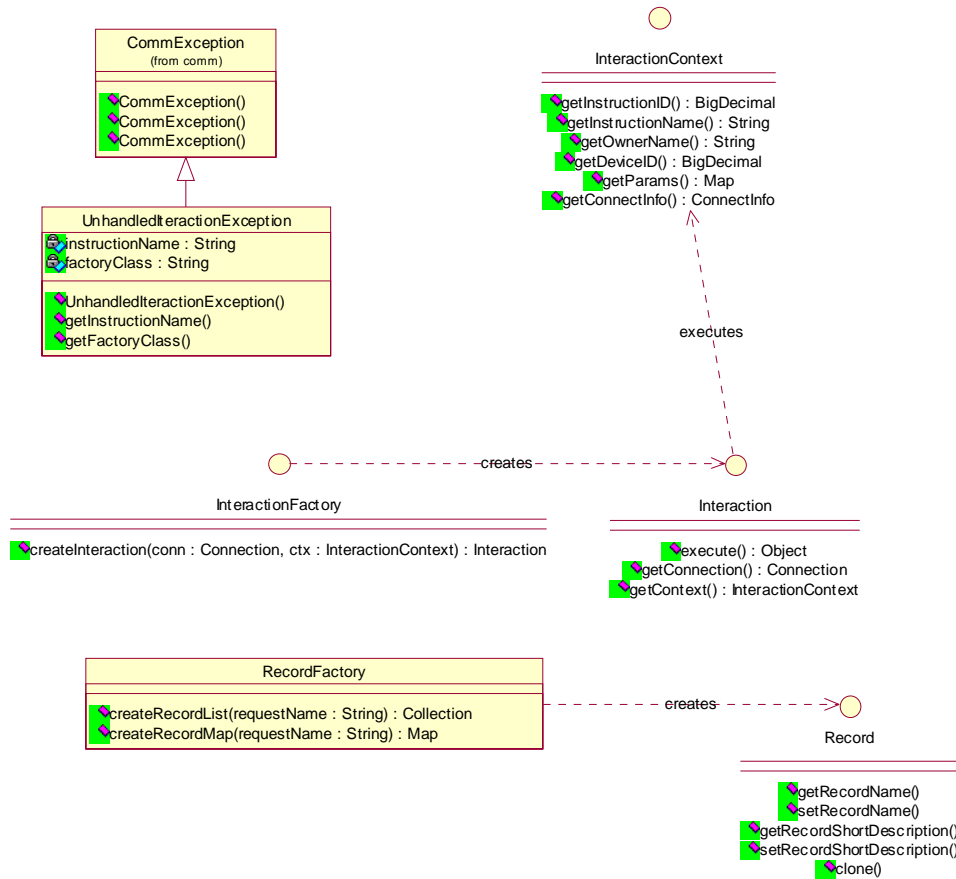


Figure 2: Comm API – Common Client Interface Class Diagram

3.1.2.2 Cdot.ctms.layer.services.comm.cci.InteractionContext

The InteractionContext maintains all of the contextual information necessary for an Interaction to perform its job. The Container passes it into the Interaction when it is invoked. The Interaction retrieves information from the context and uses it to formulate its request processing on its device. For instance, if the device is a DMS and the Interaction is for changing the message on the sign, the InteractionContext would contain information such as the message itself, the message’s priority, and how long the message should remain on the sign.

Implemented By: CTMS

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

3.1.2.3 Cdot.ctms.layer.services.comm.cci.Interaction

An Interaction is implemented for each operation that can be applied to a vendor's device. It is the "command" in the Command pattern. The implementation can hold state since the InteractionFactory must return a new instance each time. This makes the programming model easier for development as well. It is the responsibility of the Interaction to send input Records to a device and receive output Records from a device. It must convert these Records to make them meaningful to the application.

Implemented By: Service Provider

3.1.2.4 Cdot.ctms.layer.services.comm.cci.InteractionFactory

InteractionFactory implements a Factory pattern, creating the appropriate Interaction to handle a type of request. The implementation must return a new instance of an Interaction each time a request for one is made by the Container. When a vendor's Container is deployed, the deployment descriptor is configured with implementation class of the InteractionFactory that defines what interactions its devices support (see Adding Vendor-specific Extensions).

Implemented By: Service Provider

3.1.2.5 Cdot.ctms.layer.services.comm.cci.Record

Records are categorized into two types: input and output. Input Records contain data that is sent to a device while output Records contain the returned results, if any, from an operation on a device. They serve a similar purpose as java.sql.ResultSet and javax.resource.cci.Record and they are processed similarly over a "Connection" to a device.

The Record implementation can be custom or generic like a HashMap. Records are not shared across interactions. They are localized to the Interaction that is using them and discarded afterwards. They can be grouped together in an indexed collection or a map keyed by the name. The grouping depends on the type of interaction.

Implemented By: CTMS provides default for NTCIP devices otherwise the Service Provider must provide a Record implementation.

3.1.2.6 Cdot.ctms.layer.services.comm.cci.RecordFactory

A RecordFactory returns the input records to be executed on a device. If they are returned in an ordered collection, the Interaction must execute them in the order specified. If they are returned in a map, the Interaction uses the keyed names to get and execute each Record in any order as it sees fit.

CTMS has a default NTCIP RecordFactory that loads groupings of Records from an XML file called mib.xml. NTCIP compliant Service Providers must implement to provide appropriate groupings of Records for each type of Interaction supported in mib.xml. If not NTCIP compliant, the Service Provider is responsible for implementing the interface.

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

Implemented By: CTMS

3.1.2.7 `cdot.ctms.layer.services.comm.cci.UnhandledInteractionException`

This exception is specialized `CommException` that is thrown from an `InteractionFactory` when a request is made to create an `Interaction` that it does not know how to create.

3.1.3 Service Provider Interface

Classes in the `cdot.ctms.layer.services.comm.spi` package represent the Service Provider Interface to a vendor's devices. While the CCI classes represent "what" you can do to a device, the SPI classes represent "how" you do it. They are responsible for logic to open and close Connections and for encoding/decoding data across a Connection.

Once again, rather than re-inventing the wheel, the SPI classes borrow from the J2EE paradigm of connections and connection factories as well as the connection event notification mechanisms of JDBC and J2CA.

The classes in this package include: `ConfigurationException`, `ConnectInfo`, `Connection`, `ConnectionEvent`, `ConnectionEventListener`, `ConnectionFactory`, and `PooledConnectInfo`.

3.1.3.1 Service Provider Interface – Class Diagram

The following diagram shows the classes and methods in the `cdot.ctms.layer.services.comm.spi` package.

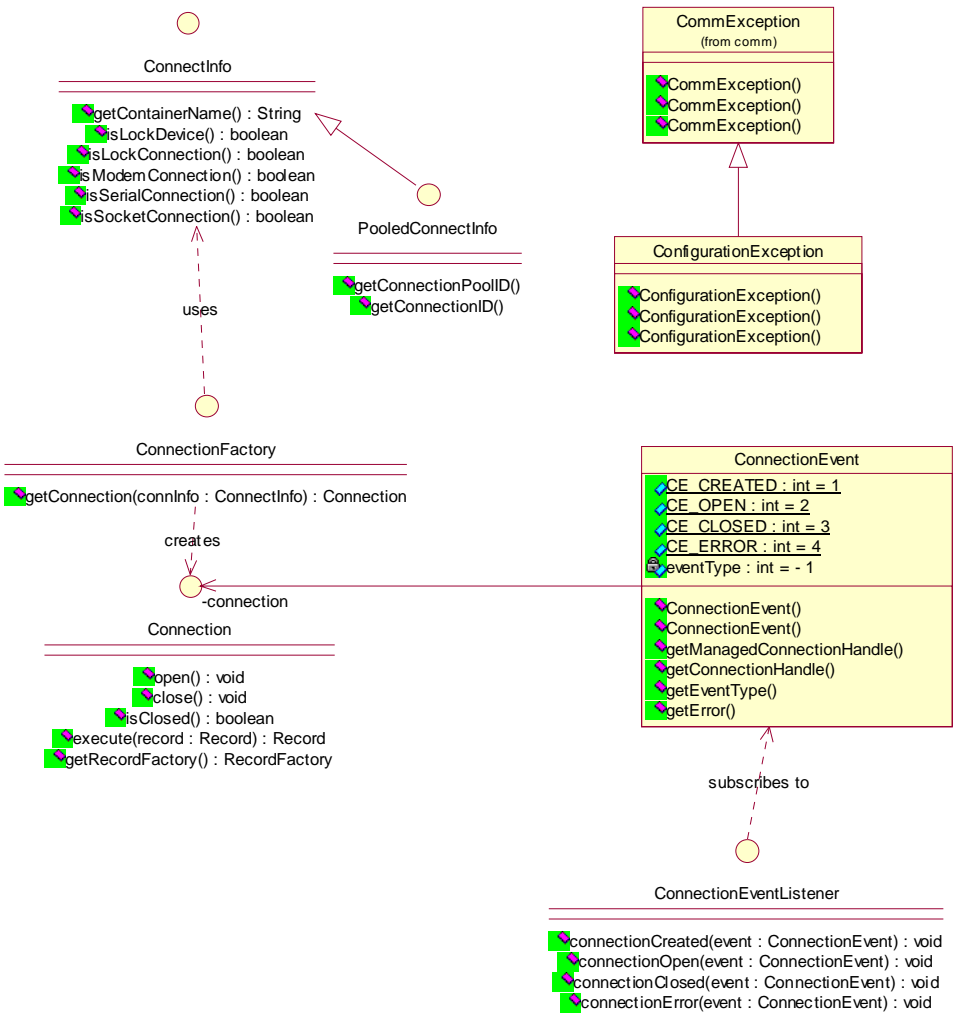


Figure 3: Comm API - Service Provider Interface Class Diagram

3.1.3.2 cdot.ctms.layer.services.comm.spi.ConfigurationException

This exception is thrown when the Comm Server encounters an error in the configuration data for a device that will prevent communications to it.

3.1.3.3 cdot.ctms.layer.services.comm.spi.ConnectInfo

ConnectInfo holds information required to connect to a device. It is passed into the ConnectionFactory.getConnection() method each time a connection is made. To help draw

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

familiarity to the J2CA specification API, ConnectInfo is similar to the javax.resource.cci.ConnectSpec and javax.resource.spi.ConnectionRequestInfor interfaces in purpose and design. In CTMS, an example of this would be a phone number to dial-up a modem-based device. Other examples include attributes like baud rate, data bits, flow control, and parity. ConnectInfo allows numerous devices to share the same physical connection (e.g., a modem port on a modem bank), since only the data (e.g., phone number) to make the connection is a different per device.

ConnectInfo is implemented in CTMS by the following classes in the cdot.ctms.layer.services.comm.connection package: CTMSSerialConnectionSpec, CTMSSocketConnectionSpec, and CTMSModemConnectionSpec. ConnectInfo data is generally static. Therefore, the CTMSContainerManager (see below) loads all configuration data into these ConnectInfo objects at startup and caches them for in-memory access by the Container when it is connecting to a device.

Implemented By: CTMS

3.1.3.4 cdot.ctms.layer.services.comm.spi.Connection

Connection provides an easily understood, generic wrapper of a physical connection to a device. Its primary responsibility is to send and receive data to and from a device. Service providers implement the Connection interface to handle protocol encoding and decoding necessary to perform these tasks against their devices. To help with the implementation, CTMS provides utilities for encoding/decoding multiple protocols (see Protocols section below).

It has a familiar API to java.sql.Connection and javax.resource.Connection including methods like: close(), executeRecord(), isClosed(), and open(). It also has a method that returns a handle to the RecordFactory, which can create Record objects that the Connection understands.

Implemented By: Service Provider

3.1.3.5 cdot.ctms.layer.services.comm.spi.ConnectionEvent

ConnectionEvents are used in the Comm Server to notify listeners about the lifecycle of a Connection. The Comm API's ConnectionEvent is similar to javax.sql.ConnectionEvent and javax.resource.spi.ConnectionEvent. There are different types of ConnectionEvents such as CE_CREATED, CE_OPEN, CE_CLOSED, and CE_ERROR. A successful interaction will include creating, opening, and then closing the connection. If a failure occurs while a ConnectionFactory is trying to open or close a Connection, the CE_ERROR event should be raised.

Implemented By: CTMS

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

3.1.3.6 `cdot.ctms.layer.services.comm.spi.ConnectionEventListener`

Any class that wants to register interest in ConnectionEvents must implement this interface. The only class that currently registers for ConnectionEvent notifications is the CTMSContainer. In this release there is no API to register custom ConnectionEvent listeners.

Implemented By: CTMS

3.1.3.7 `cdot.ctms.layer.services.comm.spi.ConnectionFactory`

The use of “connection factories” to return handles to resources is a common pattern in J2EE. It is used in EJB (`javax.ejb.EJBHome` is a factory pattern for creating EJB references), in JMS (`javax.jms.TopicConnectionFactory` and `javax.jms.QueueConnectionFactory`) and in J2CA (`javax.resource.cci.ConnectionFactory`).

Similar to these standards, a Comm API ConnectionFactory is responsible for returning a valid Connection to a device from its `getConnection()` method. Thus, the ConnectionFactory must create and open a connection or throw an exception if a valid Connection could not be made.

The Container registers the SP’s implementation of a ConnectionFactory in JNDI when it is deployed. To get a Connection, the caller performs a JNDI lookup on the ConnectionFactory that is responsible for creating the type of Connection needed. The caller then issues a `getConnection()` call passing in a `ConnectInfo` object. When a valid Connection is passed it is passed from the caller into the Interaction to process the request.

Implemented By: Service Provider

3.1.3.8 `cdot.ctms.layer.services.comm.spi.PooledConnectInfo`

The ConnectionFactory uses this interface when connecting to a device from a pool of shared connections or “comm pool”. The API contains two methods that help with identifying the pooled Connection uniquely so the factory can manage it properly. For instance, the ConnectionFactory might call `PooledConnectInfo.getConnectionID()` to get a unique identifier for the Connection object in order to determine if it is available or already in use by another device.

The CTMS Container provides a default implementation of `PooledConnectInfo` for its Connection pooling and locking. It stores Connection and Connection Pool information in configuration tables.

Implemented By: CTMS

3.2 CTMS Container

The Container is responsible for providing a pluggable platform for device integration, deployment, and execution. It ties the Comm API into a useable platform by providing an

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

runtime services to enable scalable communications to devices. Figure 4 below is a diagram of the runtime services the Container provides Service Providers out-of-the-box.

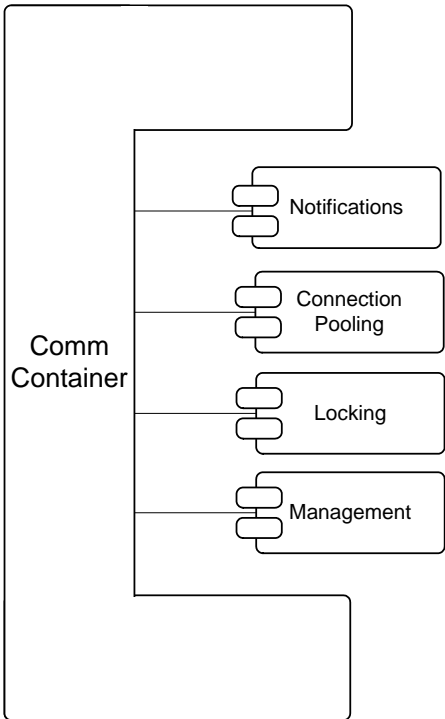


Figure 4: Container Services

3.2.1 Notifications

The Container provides notifications to inform application components about the state of an interaction. Notifications are JMX-based, so subscribers implement the `javax.management.NotificationListener` interface and register with the Container using JMX APIs. Notifications can be sent locally or to a remote interface such as a monitoring application on another machine.

3.2.2 Connection Pooling

Pooling provides a scalable approach toward connecting to devices in the communications platform. The Container implements this service so the Service Provider does not have to build pooling logic for each type of device integrated into the Comm Server. Logically, only devices with modems and IP-based communications can be pooled. This is because fiber-based devices can only be connected to a single communications port. However, the Container supports pooling fiber-based Connections as well, they just have to have one connection in the pool. CTMS actually uses pools fiber-based device Connections to standardize on the setup and implementation of Connections and Connection Pools across modems, fiber, and IP.

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

3.2.3 Locking

The Container provides defaults for locking both Connections and devices. Typically, a locking is set to true because neither the device nor the Connection can support multiple requests from the Comm Server at the same time.

3.2.4 Management

The Container is implemented using JMX; thus, it can be managed from any JMX-enabled console. With this API, you can add or remove notification listeners, start and stop the Container and modify the various configurable settings for the Service Provider's deployment.

3.2.5 Container Execution Environment

All communications to devices occur from within the Container's execution environment. The following diagram shows at a high-level how the execution environment is implemented in the Container. The main classes in the execution environment are detailed below as well.

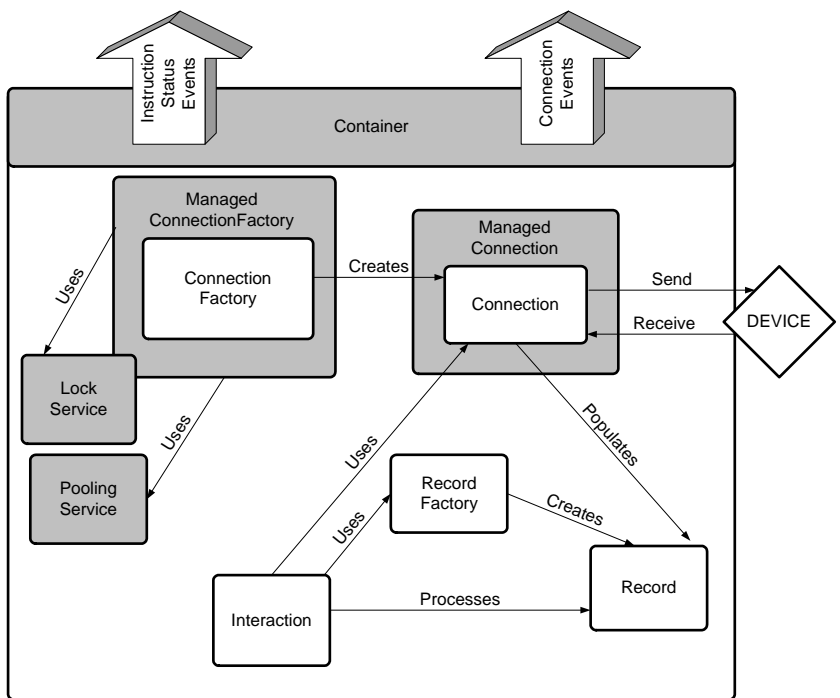


Figure 5: Container Execution Environment

In this diagram, Container provided services are shown as gray boxes while Service Provider classes are white boxes. The Service Provider classes should look familiar from the discussions above on the Comm API. The Container wraps the Service Provider implementations of ConnectionFactory and Connection to provide management capabilities and visibility into the

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

execution of an instruction to a device. This allows the Container to intercept and trap requests and events in the Service Provider classes. It also provides the ability to uniformly apply the Container services mentioned above such as locking and pooling. In essence, the Service Provider gets these for free rather than having to implement them.

3.2.6 Container

The Container includes interfaces and default implementations that manage the configuration, deployment, and execution of communications requests. As is the case with all Comm Server components, the interface's default implementation is prefixed with "CTMS". They are detailed below.

3.2.6.1 Container Class Diagrams

3.2.6.2 Container Interfaces

Figure 6 shows the Container interfaces. CTMS provides default implementations of all interfaces in the Container (see Figure 7 below), but if the default is not sufficient, another implementation can be plugged in easily (see Extending the Comm Server below).

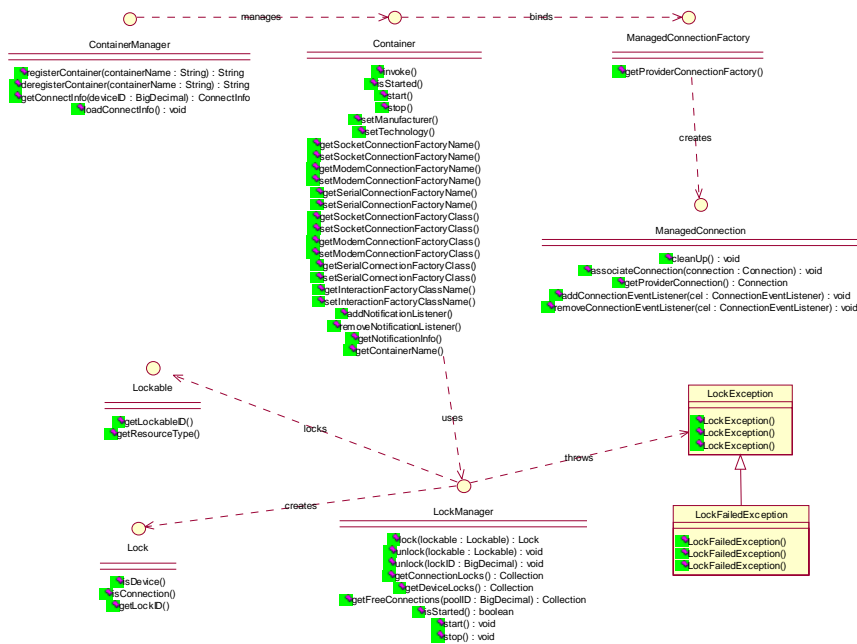


Figure 6: Container Interfaces Class Diagram

3.2.6.3 Container Implementation

Figure 7 details the default implementation of the Container interface. The classes are discussed in detail below.

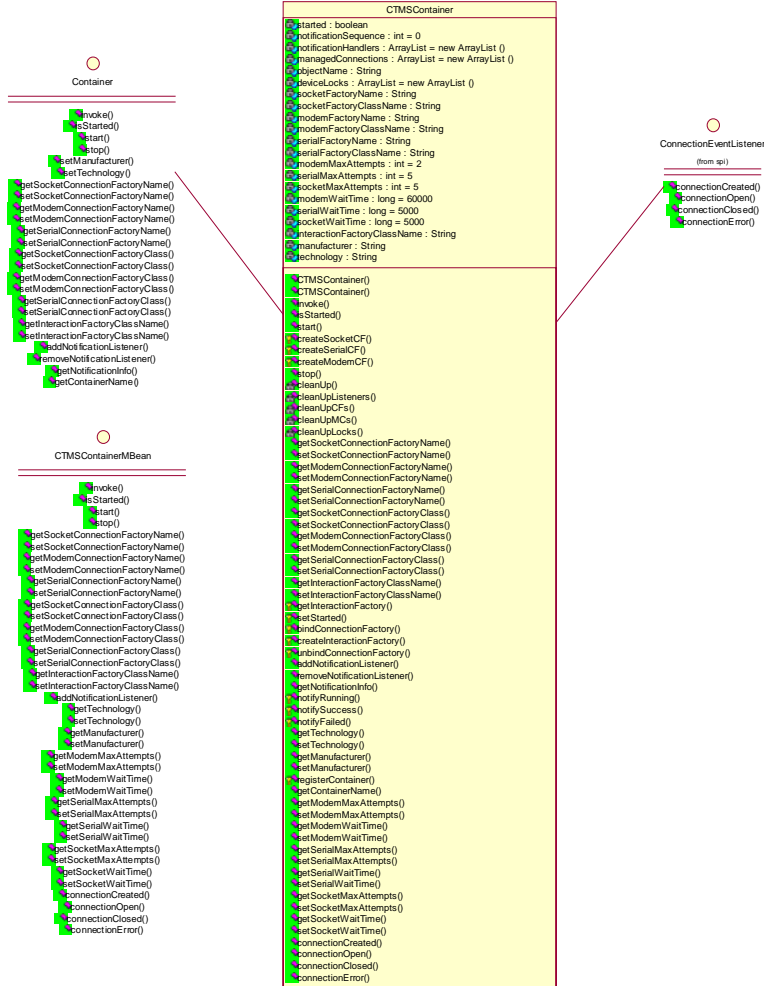


Figure 7: Container Implementation Class Diagram

3.2.6.4 Cdot.ctms.layer.services.comm.container.Container

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

The Container's responsibilities have already been discussed above. They are to provide deployment, configuration, and execution environment for communications requests. The Container implementation should provide all the services detailed above in the Container Execution Environment. For instance, Service Provider implementations can expect that Connections will be opened and closed properly no matter what happens during the processing of an Interaction. They can expect that concurrency on the Connection and device will be handled properly as well.

There are multiple Containers deployed in the Comm Server at any given time. The Container provides two methods that uniquely identify the requests it can process. They are configurable attributes on the Container: manufacturer and technology. For example, if the manufacturer, Acme, makes DMSes (e.g., technology), the Comm Server will deploy a Container that knows how to execute requests to Acme's DMSes. The same Container will not execute requests for DMSes made by another company, nor devices of another technology made by Acme.

Methods to configure a Container's manufacturer and technology are:

```
getManufacturer()
setManufacturer()
```

```
getTechnology()
setTechnology()
```

Together, the two attributes must be uniquely identifiable in the Comm Server.

The Container has configurable properties for three types of Connections. They are modem, serial, and socket, where serial represents a fiber-based device. The Container provides four configurable methods for each type of connection. They allow you to configure the name of the class that implements the ConnectionFactory interface (see Comm API), the JNDI name that the Container uses to lookup the ConnectionFactory implementation, the maximum number of attempts to connection, and the wait time between each connect attempt.

These methods are per type of Connection (e.g., modem, serial, and IP) because each type of Connection can behave differently. For instance, a SerialConnection (e.g., fiber-based) typically has a shorter execution cycle than a ModemConnection. Thus, the wait time between retries can be shorter because if the Connection is busy when first attempting to connect, it is likely that it will be released soon.

The methods for configuring Connections are (where CONNECTION_TYPE is either Modem, Serial, or Socket):

```
get[CONNECTION_TYPE]ConnectionFactoryClass()
set[CONNECTION_TYPE]ConnectionFactoryClass()
get[CONNECTION_TYPE] ConnectionFactoryName()
set[CONNECTION_TYPE] ConnectionFactoryName()
get[CONNECTION_TYPE]MaxAttempts()
```

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

```

set[CONNECTION_TYPE]MaxAttempts()
get[CONNECTION_TYPE]WaitTime()
set[CONNECTION_TYPE]WaitTime()

```

The Container interface provides a way to configure it, but provides no details or rules on when and how to actually do the configuration and deployment of the Container. This is left to the implementation.

3.2.6.5 Cdot.ctms.layer.services.comm.container.CTMSContainer

CTMSContainer is the default implementation of the Container. It implements the Container as a JMX MBean. This has two advantages to implementing Container with a regular class. They are declarative configuration using a ModelMBean deployment descriptor and manageability of the service at runtime.

First, one of the advantages of using JMX to implement the Container is that it can be configured with a ModelMBean deployment descriptor. As noted earlier, the Container interface provides no rules for when and how configuration and deployment occur. A CTMSContainer is deployed when its deployment descriptor is deployed in the MBean Server (e.g., JBoss Server). JBoss Server recognizes the deployment descriptor and automatically deploys the Container into the MbeanServer. During this process, it uses values defined in the deployment descriptor to configure the CTMSContainer by calling the appropriate mutator methods on the Container interface (e.g., setModemConnectionFactoryName()). Once all of the attributes are initialized, the start() method is called to start the CTMSContainer service. At this time, CTMSContainer will use the configured properties to create and bind the ConnectionFactory classes to JNDI; thereby, preparing it to handle requests for devices.

Second, another advantage of implementing the Container as a JMX MBean is that it can be managed from any JMX-enabled console, providing visibility into the service for monitoring and service re-configuration, starting and stopping.

Besides Container, CTMSContainer also implements the ConnectionEventListener interface, so it can receive feedback on ConnectionEvents during the processing of an instruction.

3.2.6.6 Cdot.ctms.layer.services.comm.container.ContainerManager

The ContainerManager is responsible for loading and reloading device configuration information, so Containers have this information available when managing the device communication. It is also responsible for managing Containers in the Comm Server. Thus, each Container is registered with the ContainerManager when it is deployed and de-registered when it is undeployed. The ContainerManager provides the ability to lookup a Container based on the device ID. The Comm Server uses this to get the proper Container to execute an instruction.

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

3.2.6.7 Cdot.ctms.layer.services.comm.container.CTMSContainerManager

The CTMSContainerManager is the default implementation of the ContainerManager interface. It is a JMX MBean exposing all ContainerManager methods as manageable JMX APIs. The CTMSContainerManager loads device configuration information from the following tables:

DEVICE_CONNECT_SPEC
 DEVICE_IP
 DEVICE_MODEM
 DEVICE_SERIAL

DEVICE_CONNECT_SPEC is common across all devices and contains information like the type of protocol used and whether or not to lock the device and connection during processing. The other tables have connection specific data elements that are required for each type of connection. For instance, DEVICE_MODEM contains a phone number to dial when contacting a modem-based device. For more information on these tables, please see the database design document.

Device configurations are loaded at start of the MBean and they are cached and reloaded as configuration data changes in the application. They do not change often. Thus, rather than fetch configuration data each time a device needs to be contacted, caching provides a fast, in-memory lookup that is much more scalable.

3.2.7 Connection Management

The following section describes the components involved in the connection management services provided by the Container. Connections are created using the Factory pattern that is standard in J2EE. In this pattern, a ConnectionFactory is bound in JNDI during deployment. It is accessed by the application using a JNDI lookup name and then called to create an instance of a Connection object to a specified resource. The Comm Container implements this familiar paradigm as well. The classes that collaborate in this pattern are detailed below.

3.2.7.1 Managed Connection Factory Class Diagram

Figure 8 shows the relationship between the SPI and Container classes for managing connections.

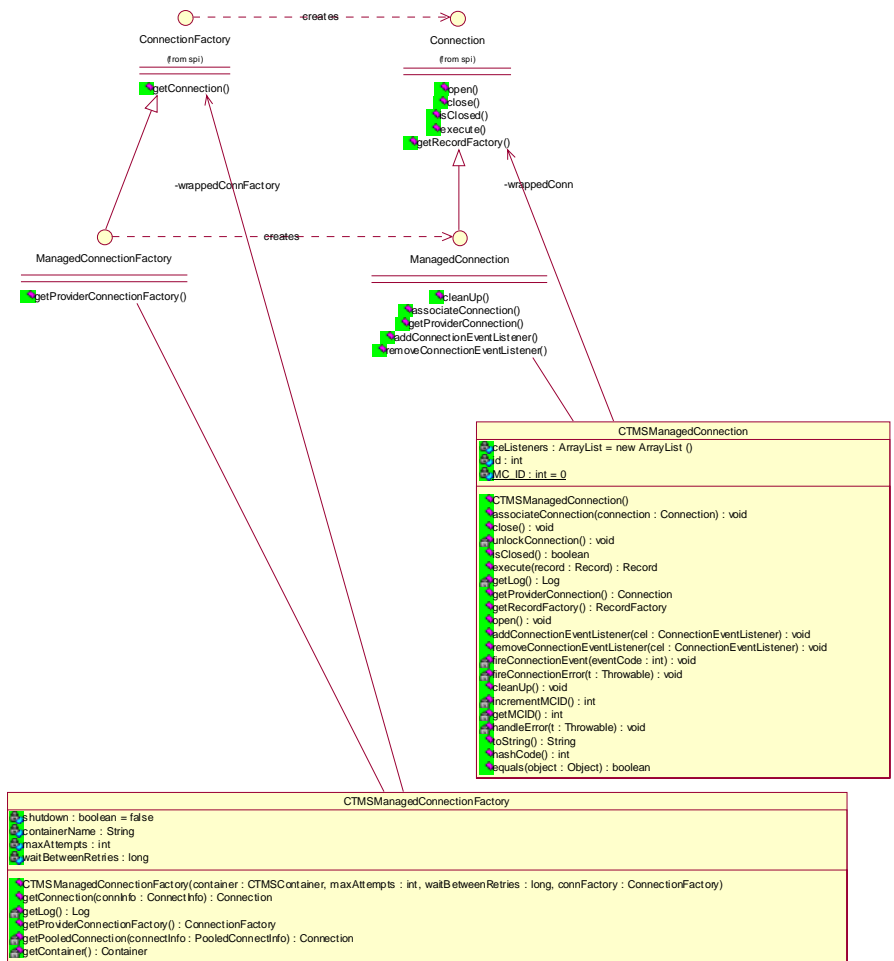


Figure 8: CTMSManagedConnection Class Diagram

3.2.7.2 Managed Connection Factory Sequence Diagrams

The following sequence diagrams show the steps involved in successfully connecting to a device via a ModemConnection. The Service Provider in the example is an actual CTMS supported vendor, Skyline. The sequence diagrams are broken into two: Get Connection 1 and Get Connection 2. The second is nested inside the first at the spot indicated by an attached note in the diagram.

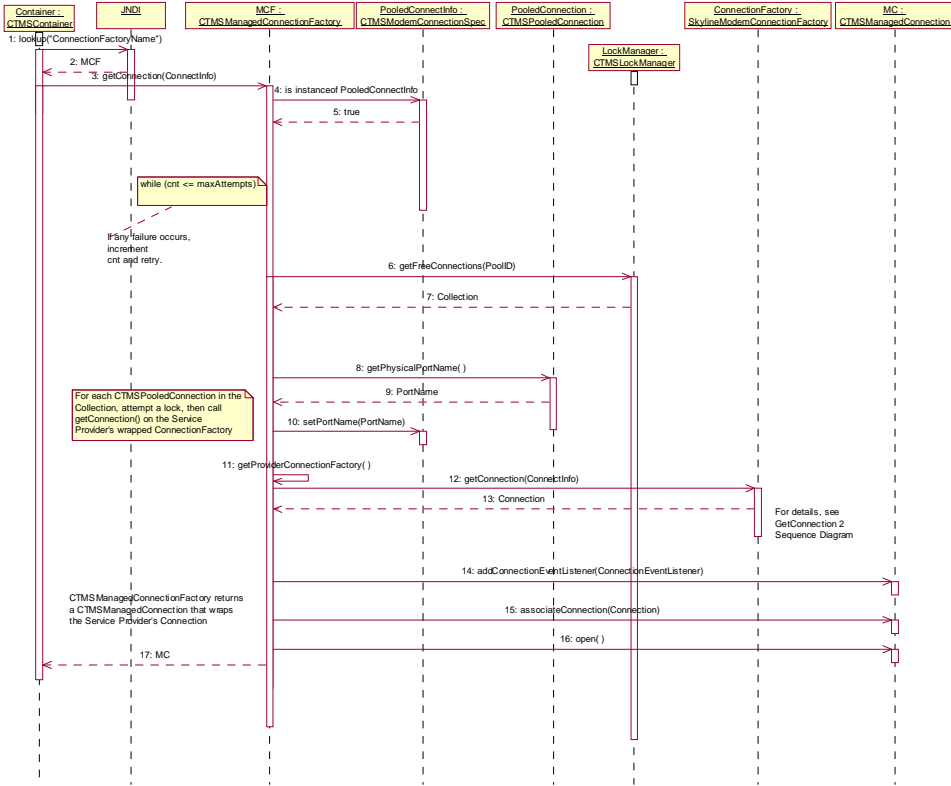


Figure 9: Get Connection 1 - Sequence Diagram

Figure 9 shows the steps taken by the Container to perform Connection pooling services to return a valid Connection object for Interactions to use when communicating with a device.

First, the Container looks up the ConnectionFactory in JNDI and the steps performed by the wrapper class, CTMSManagedConnectionFactory (MCF), to establish a Connection. The MCF determines if the ConnectInfo object passed to getConnection(ConnectInfo) is for a pooled or non-pooled connection. In this case, the ConnectInfo is actually of type “PooledConnectInfo” from the Comm API, so the MCF invokes a special method on itself for handling connection pooling called getPooledConnection(PooledConnectInfo). In this method, the MCF uses the LockManager to determine which Connections in the pool are free (e.g., not locked) by calling LockManager.getFreeConnections(poolID). As noted in the CTMSLockManager description, this method looks to make sure the Connection ID is not in the COMM_LOCK table. If it is not, it will proceed with an attempt to lock it. Figure 9 assumes a successful lock. The MCF then adds the physical Connection information such as the physical port number to the PooledConnectInfo object and calls the wrapped Service Provider ConnectionFactory (e.g.,

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

SkylineModemConnectionFactory) to create the appropriate Connection. See Figure 10: Get Connection 2 for more details on the sequence of events.

After a valid Connection is returned from the SkylineModemConnectionFactory, the MCF creates a new CTMSManagedConnection (MC) to wrap the Service Provider's Connection object. Then, adds the Container as a ConnectionEventListener, so it is registered for notifications on lifecycle events of the Connection. The Service Provider doesn't have to do anything to get this capability because the MC wrapper will perform all ConnectionEvent notifications for it automatically. Once the Container is listening, the MC.open() method is invoked. This delegates to the wrapped Connection's open() method to perform specific logic to open a physical CommPort. If a failure occurs here, the Connection will be unlocked and destroyed (see MC.cleanUp()). In the event of a failure, the MCF will wait for a specified amount of time and then it will attempt to connect as many times as it was configured in the deployment descriptor.

Finally, the MC is returned to the Container. Since the MC also implements the Comm API's Connection interface, polymorphism tricks the caller into thinking it is working with a regular Connection, not a wrapper. Interactions should not downcast this MC instance, thinking it is the Service Provider's Connection because a ClassCastException will be thrown.

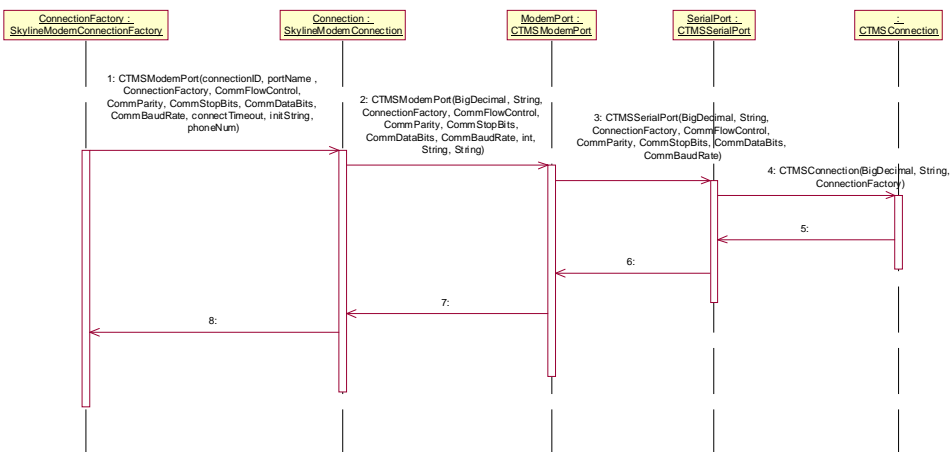


Figure 10: Get Connection 2 - Sequence Diagram

Figure 10 shows how a Service Provider's ConnectionFactory creates and returns a Connection instance. In this case, the SkylineModemConnection takes advantage of base classes provided by CTMS that know how to perform communications of CommPorts. Relevant connection data is passed in the constructor including references to the ConnectionFactory instance that created the Connection. Note: The constructors do not attempt to connect to the physical ports at this time. The Connection.open() method is used at a later time.

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

3.2.7.3 Cdot.ctms.layer.services.comm.ManagedConnection

ManagedConnection (MC) is a wrapper around the Service Provider's Connection implementation. It provides ConnectionEvent services, notifying listeners of interesting ConnectionEvents such as when the connection was created, opened, and closed. MC inherits from Connection in the SPI package, so it looks like a regular Connection to the application. However, it has extra management functionality exposed in the MC interface to allow the Container to perform tasks such as associating the MC with a Service Provider Connection or cleaning up the MC's resources when it is in a completed or failed state.

The cleanup() method should destroy the MC and any physical handles to underlying resources that it maintained. It is called from the Container when a CE_ERROR ConnectionEvent is caught or when the Container is shutting down. It must never throw an exception or it may interrupt the Container's processing.

The associateConnection() method sets the reference to the underlying, physical Connection that the MC manages. This method should produce a CE_CREATED ConnectionEvent to notify the Container that a new Connection has been created.

The getProviderConnection() method returns the underlying, physical Connection implemented by the Service Provider. This will return null if called before associateConnection() is called.

Finally, add/removeNotificationListener() are used to managed ConnectionEventListener object registration.

3.2.7.4 Cdot.ctms.layer.services.comm.CTMSManagedConnection

CTMSManagedConnection is the default implementation of the MC interface methods as specified above. It provides intercepting and delegation behaviors and tracks and notifies ConnectionEventListener objects. The CTMSManagedConnection keeps a unique ID called an MC_ID purely for tracking purposes. It is only unique within a Comm Server instance; thus, not across a cluster of Comm Servers. An example usage of MC_ID would be so the ConnectionEventListeners can uniquely identify which MC emitted a notification. The logging framework also takes advantage of the MC_ID, which can be helpful in tracking errors in the logs.

CTMSManagedConnection uses a helper method, unlockConnection(), to call the LockManager implementation and unlock the managed Connection object. It is called from close() (e.g., Connection.close()) and cleanUp(e.g., ManagedConnection.cleanUp()) to make sure no resources are left lock in the event of a completed or failed Connection.

3.2.7.5 Cdot.ctms.layer.services.comm.ManagedConnectionFactory

The ManagedConnectionFactory interface (Figure 8) is used by the Container to provide management APIs. It currently has only one public method, getProviderConnectionFactory(), which returns the wrapped ConnectionFactory instance. The Container can use this to access the Service Provider's ConnectionFactory directly, if need be.

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

3.2.7.6 Cdot.ctms.layer.services.comm.CTMSManagedConnectionFactory

Figure 8 shows the default implementation of the Container's ManagedConnectionFactory interface. It provides several important Container services such as locking and Connection pooling. It is serializable, so it can be bound in JNDI. When binding the ConnectionFactory instances defined in the deployment descriptor, the Container will wrap each with an instance of CTMSManagedConnectionFactory to provide call interception to the wrapped ConnectionFactory.

3.2.8 Connection Spec Classes

This section describes the default CTMS implementations of the Comm API's ConnectInfo interface. The classes reside in the Cdot.ctms.layer.services.comm.connection package. They contain attributes necessary for the ConnectionFactory to create a valid Connection. They are loaded and cached by CTMSContainerManager when the service starts. When requests for a particular device are made, the ContainerManager determines which ConnectInfo object to use to connect to the device and returns it. For a detailed class diagram of these classes see Figure 11 below.

Note: Currently, these classes are used as-is by the Comm Server and cannot be extended without also requiring changes to the classes that access and load them like CTMSContainerManager and ContainerExchange (see Extending the Comm Server below). Thus, if a new type of Connection or a new attribute needs to be added, both classes will be affected. Future implementations may provide a Factory pattern capability to allow custom implementations to be more easily plugged in without modification to the core Comm Server.

3.2.8.1 Cdot.ctms.layer.services.comm.connection.CTMSConnectionSpec

CTMSConnectSpec is the base class for all ConnectInfo implementations. It provides basic attributes and behavior used by all subclasses. Most attributes are self-explanatory. For instance, lockDevice is a Boolean flag that indicates whether the device should be locked before contacting it. The lockConnection attribute likewise, applies to whether or not to lock the Connection object. Thus, the Container changes its behavior based on these fields. By design, all Connections in CTMS are pooled even if they are fiber and can only have a single Connection. Thus, you can see in Figure 11 that CTMSConnectionSpec implements the Comm API's PooledConnectInfo interface.

As noted earlier under CTMSContainerManager, CTMS stores all device and Connection related configuration data in a database. Administrators of the CTMS application manage this data from the CTMS application where security controls can be applied to ensure the information is not tampered with.

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

3.2.8.2 Cdot.ctms.layer.services.comm.connection.CTMSSerialConnectionSpec

The CTMSSerialConnectionSpec class provides attributes to help a ConnectionFactory understand how to create a Connection over a serial CommPort. Fields include CommDataBits, CommFlowControl, CommParity, CommStopBits, and BaudRate.

3.2.8.3 Cdot.ctms.layer.services.comm.connection.CTMSModemConnectionSpec

The CTMSModemConnectionSpec specializes the CTMSSerialConnectionSpec even more to include data fields necessary to perform dial-up over a modem.

3.2.8.4 Cdot.ctms.layer.services.comm.connection.CTMSSocketConnectionSpec

The CTMSSocketConnectionSpec provides two extra fields required to make a socket-based Connection to a device. They are the IP Address and Port number that the device's socket listener is receiving on. CTMS does not currently use this class, but has it as a placeholder for the future.

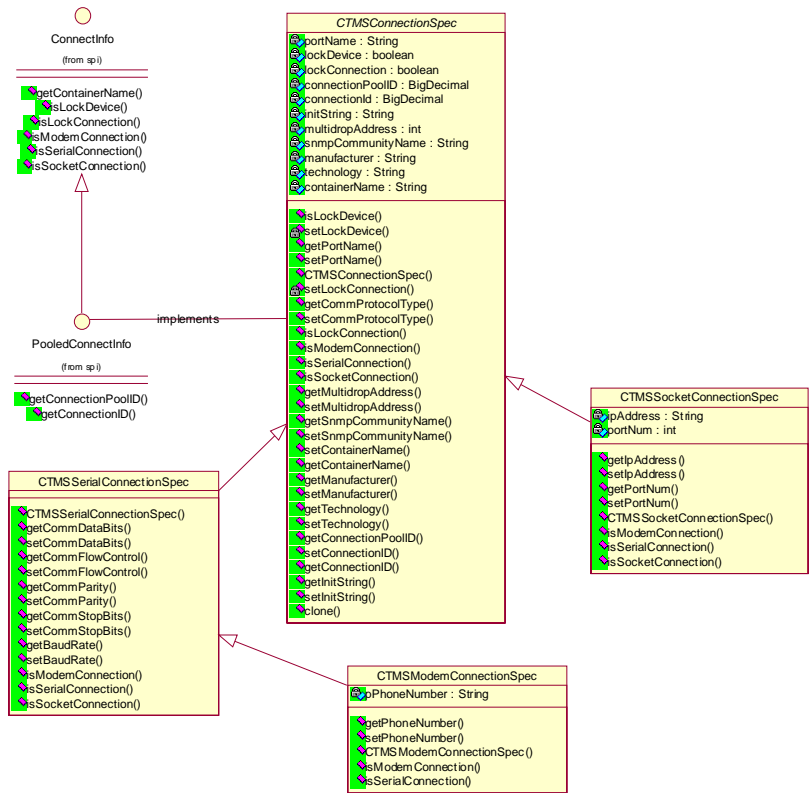


Figure 11: CTMS ConnectInfo Class Diagram

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

3.2.9 Connection Classes

CTMS provides base classes for the three Connection types: Modem, Serial, and Socket. They have a similar hierarchy to the ConnectionSpec classes. The details of these classes can be seen in Figure 12 below. Although not required, Service Providers are recommended to use these base classes when extending the Comm Server to add Connection implementations to their devices. That way future improvements to the core Comm Server can be applied to all Connection subclasses.

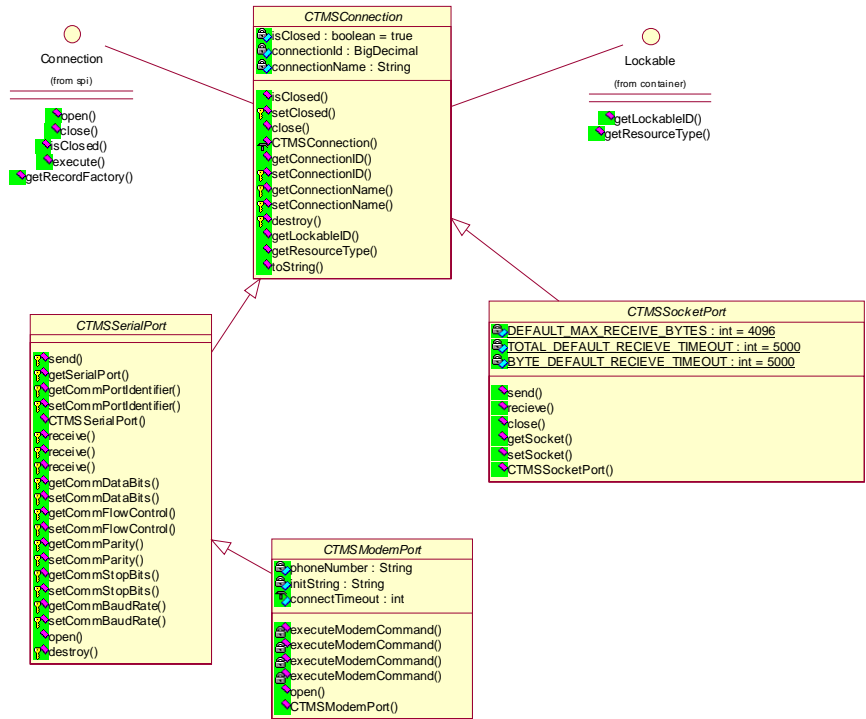


Figure 12: CTMS Connection Class Diagram

3.2.9.1 Cdot.ctms.layer.services.comm.connection.CTMSConnection

CTMSConnection implements only a few APIs from the Connection interface. It primarily adds convenience attributes and methods and implements the close() and isClosed() methods on the Connection. This class implements the Lockable interface since Connections can be locked by the LockManager service.

The close() method is a template pattern. It calls destroy() and then sets the isClosed attribute to false. Subclasses must override the abstract destroy() method to physically close the underlying Connection and clean up any left over resources. For instance, CTMSSerialPort will close the CommPort that was opened to perform serial communications.

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

The getConnectionID() method returns connectionId, the unique identifier of the Connection object as stored in the CTMS configuration table, CONNECTION.

The setConnectionID() is a mutator for the connectionId field.

The getConnectionName() method returns the physical connection or “port name”.

The setConnectionName() method is a mutator for the connectionName field.

3.2.9.2 Cdot.ctms.layer.services.comm.connection.CTMSSerialConnection

CTMSSerialConnection extends the base class to provide attributes and behavior specific to managing serial CommPorts. It uses the RXTX Comm APIs from GNU to perform serial communications. These APIs are Open Source and freely available. While they work on the Windows platform, they are highly optimized for the Linux platform; thus, CTMS chose to use them for deploying the Comm Server on Red Hat. The APIs are almost identical to the Java Comm APIs except for the package names. Thus, if required to use Java Comm APIs instead, it is a minor effort to change to import Java Comm classes.

3.2.9.3 Cdot.ctms.layer.services.comm.connection.CTMSModemConnection

CTMSModemConnection extends CTMSSerialConnection because it performs dial-up connectivity over a serial connection.

The executeModemCommand() methods are helper methods that execute commands over the connection. The list of available modem commands can be found in the constants class cdot.ctms.layer.services.comm.connection.CommModemCommands. The methods throw SendFailedException and ReceiveFailedException depending on the state of the CTMSModemConnection during the operation. If encountered, CTMS ends communications and does not try to send or receive the data again.

The open() method overrides the super class to not only open the CommPort, but to dial-up the device through the modem. It handles busy tones by throwing an exception to the ConnectionFactory if one is encountered. While device and Connection locking should prevent busy tones from within CTMS, CTMS cannot guarantee that 3rd party applications are not connecting to a device.

3.2.9.4 Cdot.ctms.layer.services.comm.connection.CTMSSocketConnection

CTMSSocketConnection represents a connection over a socket to a device. This class is a placeholder for the future. It does not provide real functionality and is not currently used.

3.2.10 Lock Manager

The following section describes the Lock Manager in more detail. For the class diagram of the LockManager, see Figure 13 below. The locking services provided by the Container include a set of interfaces, exceptions, and default implementations.

3.2.10.1 Lock Manager Class Diagram

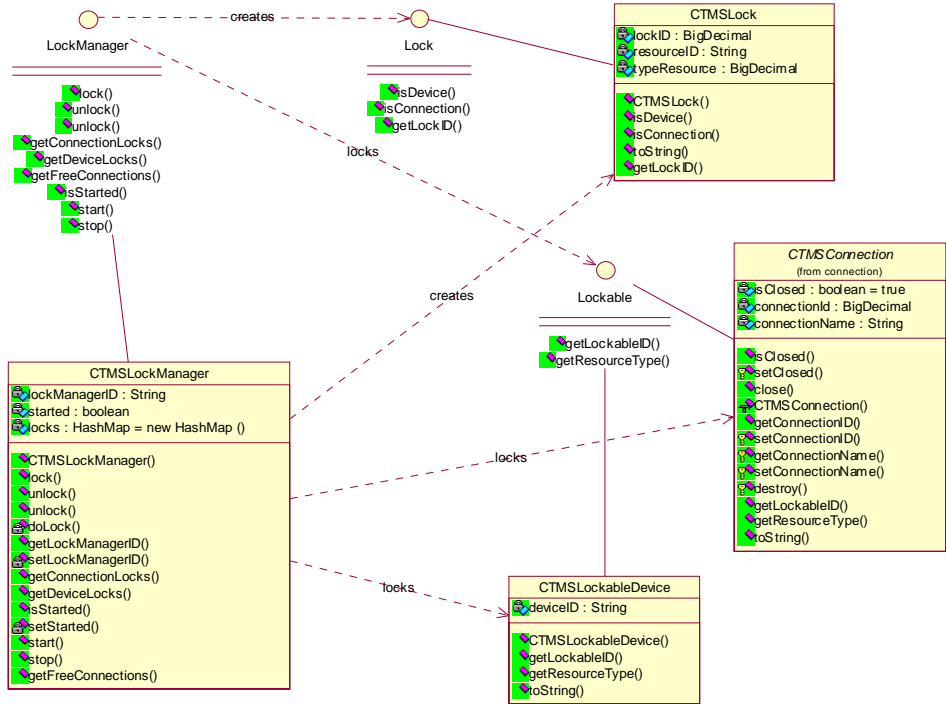


Figure 13: CTMSLockManager Class Diagram

3.2.10.2 Cdot.ctms.layer.services.comm.container.LockManager

The `cdot.ctms.layer.services.comm.container.LockManager` interface provides an API for locking and unlocking devices and connections. Actually, it works with a `Lockable` interface, so it can lock and unlock anything that implements this interface, not just Connections and devices. `CTMSContainer` uses the `LockManager` to provide locking services to Service Providers automatically. Besides `lock()` and `unlock()` methods, the `LockManager` also provides an easy way to see the locks and types of locks currently held (e.g., device or Connection). The `LockManager` is used for pooling services. There is only one `LockManager` service per Comm Server.

3.2.10.3 Cdot.ctms.layer.services.comm.container.CTMSLockManager

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

The default locking implementation, CTMSLockManager, uses a database table called DEVICE_CONNECT_SPEC to hold locking information in a distributed environment. This ensures locks are recognized across a cluster of Comm Servers. A device-specific setting in the CTMS configuration tables configures locking at the device and connection level. The columns that regulate locking are: LOCK_DEVICE_FLG and LOCK_CONNECTION_FLG. Setting these fields to 1 will ensure they are locked while 0 will tell the Container that they support multiple communications at the same time.

All LockManagers must be uniquely identified in the system in order to determine which LockManager owns a lock. The CTMSLockManager uses a System property called "lock.manager.id" as its identifier. When running clustered Comm Servers, it is important to ensure this System property is different across instances. Otherwise, the clustered CTMSLockManagers will behave abnormally by clearing and unlocking each other's locks!

The getFreeConnections() API provides the "pooling service" for the Container. It returns Connections in a pool that are free to use. If it returns zero entries, all are currently in use and the Container should wait and try again later. Essentially, a free Connection is one that is not locked or otherwise currently servicing a device.

The getDeviceLocks() method will show all device locks currently held in the CTMSLockManager. It will not return locks made by other LockManagers.

The getConnectionLocks() method will show all Connection locks currently held in the CTMSLockManager. It will not return locks made by other LockManagers.

3.2.10.4 Cdot.ctms.layer.services.comm.container.Lockable

Lockable interface represents any object that can be locked by the LockManager. It is the argument passed into both the lock() and unlock() methods of the LockManager interface. It is currently implemented by three classes in the Comm Server: CTMSLockableDevice, CTMSConnection, and CTMSPooledConnection (not shown in Figure 13 above).

A Lockable object must be uniquely identified. It is identified using a lock ID and a type of lock, or "resource type". The implementation of the interface must return values that are globally unique across the system.

3.2.10.5 Cdot.ctms.layer.services.comm.container.Lock

The Lock interface represents a successful, currently locked Lockable. The Lock will have representation in the COMM_LOCK table and a local cache in the CTMSLockManager that performed the lock. There is currently two types of locks, either a device lock or a Connection lock. CTMSLock implements this interface by default and it uses the resource type of the Lockable to return whether or not the Lock is a device or Connection.

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

3.2.10.6 Lock Manager - Lock Sequence Diagram

Figure 14 shows the sequence of steps taken to lock a device or Connection. Both implement the Lockable interface and the CTMSLockManager makes no distinction between the two.

The caller invokes the lock() method, passing in the Lockable object to lock. CTMSLockManager implements LockManager, by default. This class uses a local cache of Locks it has performed to boost performance when checking for current locks. If one is not found in the lock cache, it will check the COMM_LOCK table using the ContainerExchange.lock() method. Thus, the COMM_LOCK table ensures that locks are propagated across a cluster of Comm Servers, since each will check this shared resource during the lock process. If an attempt to lock a Lockable fails, the CTMSLockManager will retry to procedure. Not shown on the sequence diagram is a “while” looping construct that tries the doLock() method until MAX_TRIES (default of 5) is reached. If the Lockable has not become unlocked in that time, a LockFailedException is thrown to the caller.

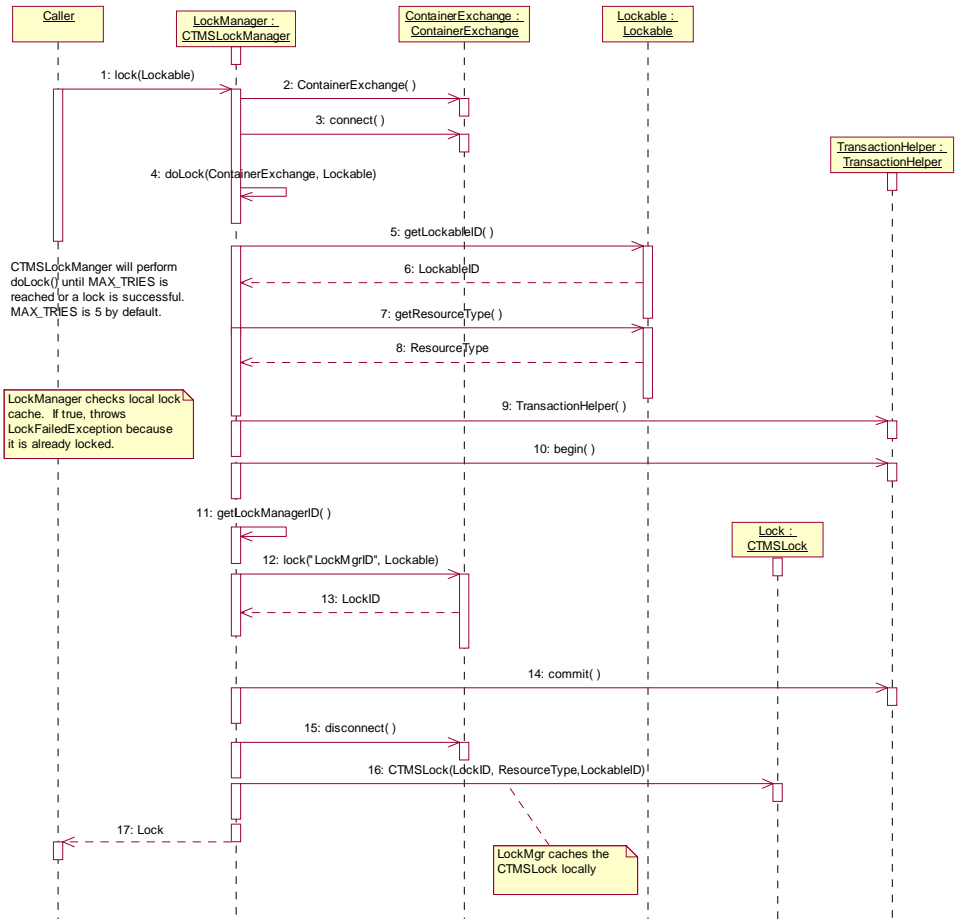


Figure 14: Lock - Sequence Diagram

3.2.10.7 Lock Manager - Unlock Sequence Diagram

Below is a sequence diagram of the steps taken by CTMSLockManager to unlock a locked object.

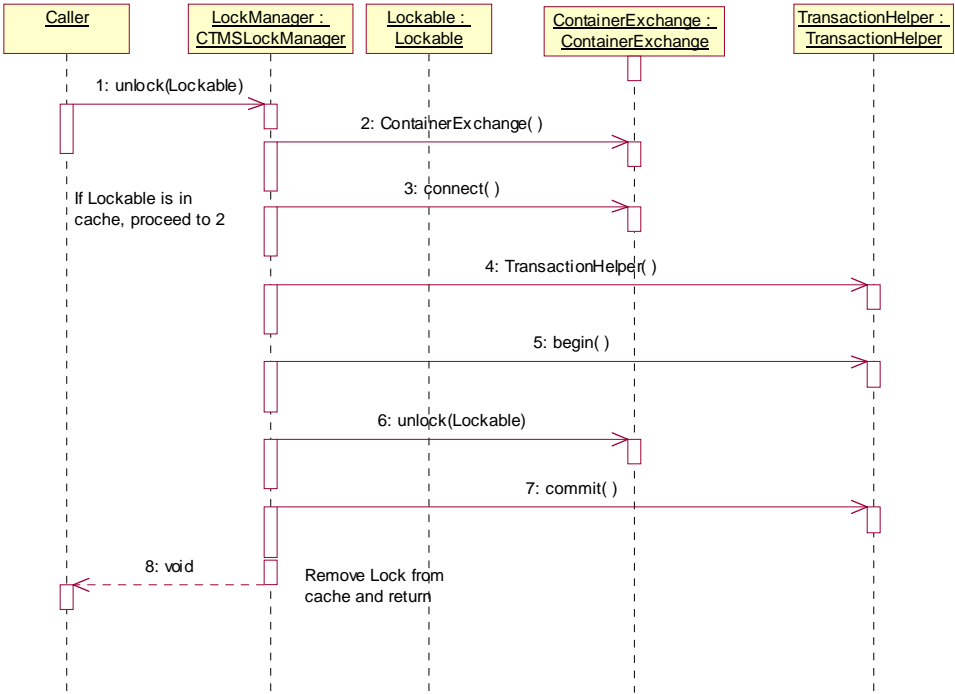


Figure 15: Unlock - Sequence Diagram

The process of unlocking an object is similar to the lock process. The caller passes in the Lockable to unlock to the CTMSLockManager. The CTMSLockManager checks to see if it owns this lock by looking in its cache of locks. If it does not exist, it exists, assuming it was already unlocked or owned by another Comm Server’s LockManager. If it exists, the ContainerExchange is created, and connected to the database. A transaction is initiated by the help of the TransactionHelper instance’s begin() method. Then, the ContainerExchange.unlock(Lockable) is invoked. This will remove the row from the COMM_LOCK table if one exists. ContainerExchange will throw a NoLockFoundException, if no rows are removed from COMM_LOCK. The LockManager can ignore this exception, since it was trying to remove the lock anyway. Finally, the CTMSLockManager will remove the Lock reference from its local cache and return.

Note: LockManager also has an unlock(BigDecimal), which takes the unique id to unlock (e.g., the LockID). The sequence of steps is similar to the unlock(Lockable) shown above, so it is not detailed here. This overload is handy if a lock is hung and needs to be unlocked from a management console. Using the getDeviceLocks() or getConnectionLocks() APIs, you can discover which lock ID is hung and then pass it into the unlock() method.

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

3.3 Protocol Adaptors

The Protocol adaptor classes take byte arrays generated by the NTCIPRecord and NTCIPGroupRecord classes and encode or decode the byte stream to/from the Open System Interconnection layer 2 protocols. Presently all the devices used by C-Dot use the point-multi-point protocol. This protocol is a more specific standard of the High Level Data Link Control specification.

3.3.1 ProtocolAdaptor

This interface defines the only two behaviors, encode() and decode(). The assumption is that all representation of OSI layer 2 protocols can be accomplished this way.

3.3.1.1 HDLCProtocolAdaptor

This class performs the encoding and decoding of the High Level Data Control specification and implements the ProtocolAdaptor interface. The encode() method replaces the hex value 0x7d with the values 0x7d, 0x5d. It then loops for each byte in the array and replaces the 0x7e with 0x7d, 0x5e values. This is because the 0x7d and 0x7e have special meaning in the communication packet for this specification.

3.3.1.2 PMPPProtocolAdaptor

This protocol adaptor extends the HDLCProtocolAdaptor and further refines the how the communication packet is constructed. The encode method adds the following fields to the communication packet:

- Multi Drop Address
- Control field
- Initial Protocol Identifier – Simple Transportation Management Framework
- Cyclic Redundancy Check 16
- Start Book End
- Stop Book End

The decode method simply removes the bookends, then decodes the HDLC escapes then removes the frame check sequence and verifies it against the CRC16 model. Then removes the multi drop address, then the control field and finally the IPI flag. The remaining the byte array is the SNMP packet that is decoded by the NTCIPRecord.

3.3.1.3 Protocol Adapter Class Diagram

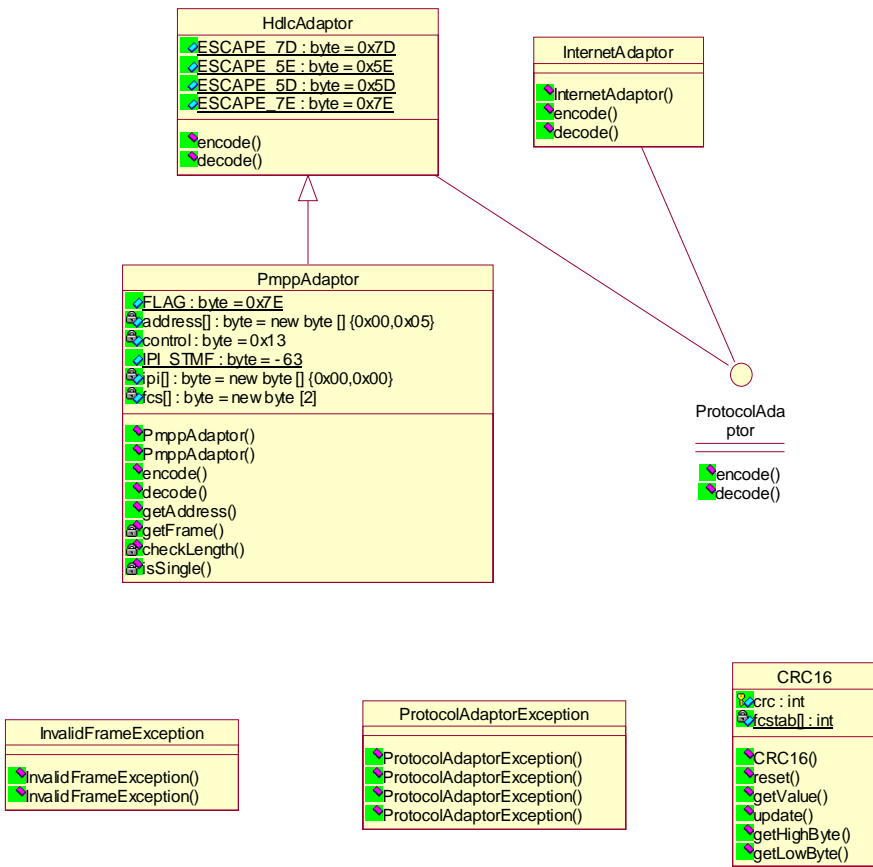


Figure 16: Protocol Adapter Class Diagram

3.3.1.4 Protocol Adapter Encode Sequence Diagram

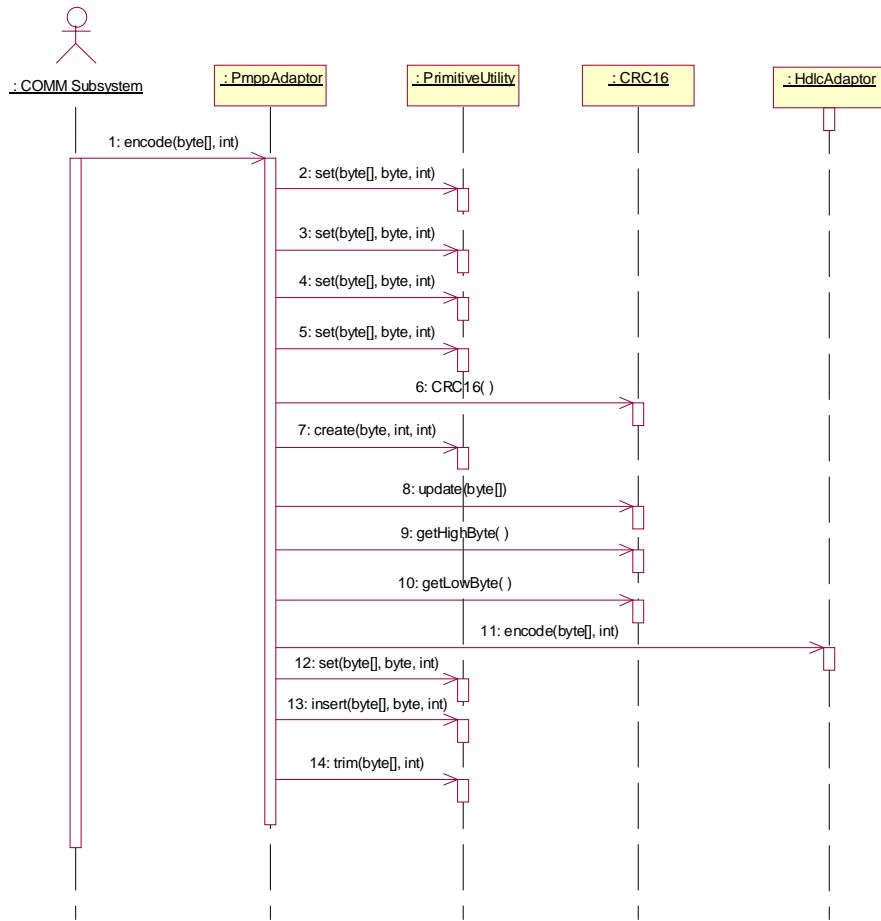


Figure 17 Protocol Adapter Encode Sequence Diagram

The encoding of the byte array from the Pmp specifications starts with a call to the PmpAdaptor. The following steps are in order:

- Add the multi-drop address
- Add the PMPP control field
- Add the initial protocol identifier
- Create a new CRC16
- Add the current array to the CRC16
- Set the Frame Check Sequence high byte
- Set the Frame Check Sequence low byte
- Encode the array with HDLC encode method

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

- Set the end book end
- Set the start book end
- Trim the array

3.3.1.5 Protocol Adaptor Decode Sequence Diagram

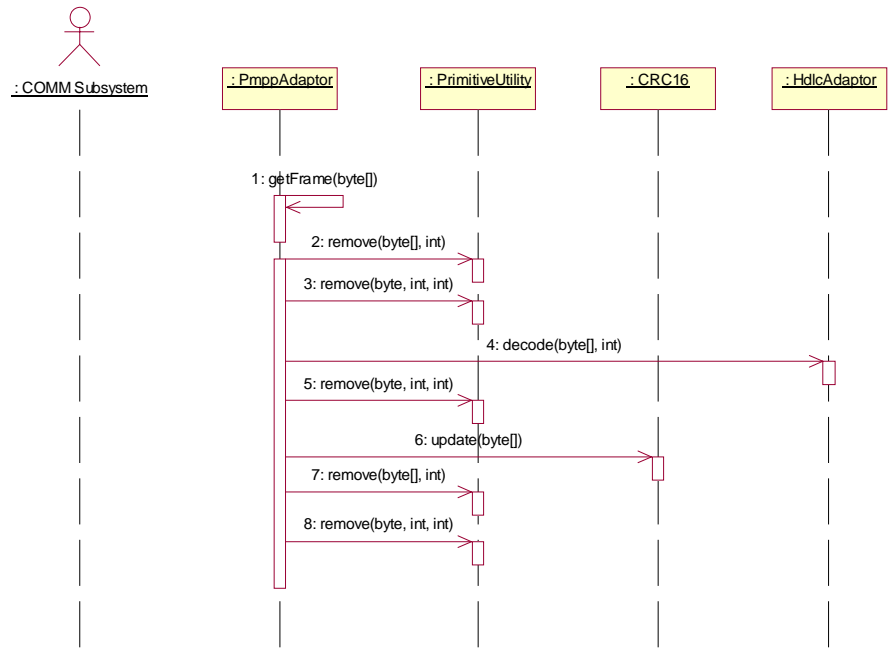


Figure 18 Protocol Adaptor Decode Sequence Diagram

The decoding process removes all fields related to the OSI layer 2 specification leaving only device communication packet. In this application the devices are using SNMP. The decoding follows these steps:

- Remove the end bookmark
- Remove the start bookmark
- Decode the array with HDLC decode method
- Get the Frame Check Sequence low byte
- Get the Frame Check Sequence high byte
- Remove them from the current frame
- Create a new CRC16
- Add the current frame to it
- Compare the CRC16 to the FCS
- Remove the multi-drop address

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

- Remove the control field

3.3.1.6 Protocol Packet Diagram

The following diagram shows the embedding of the PDU inside the SNMP packet and the embedding of the SNMP packet inside the PMPP packet.

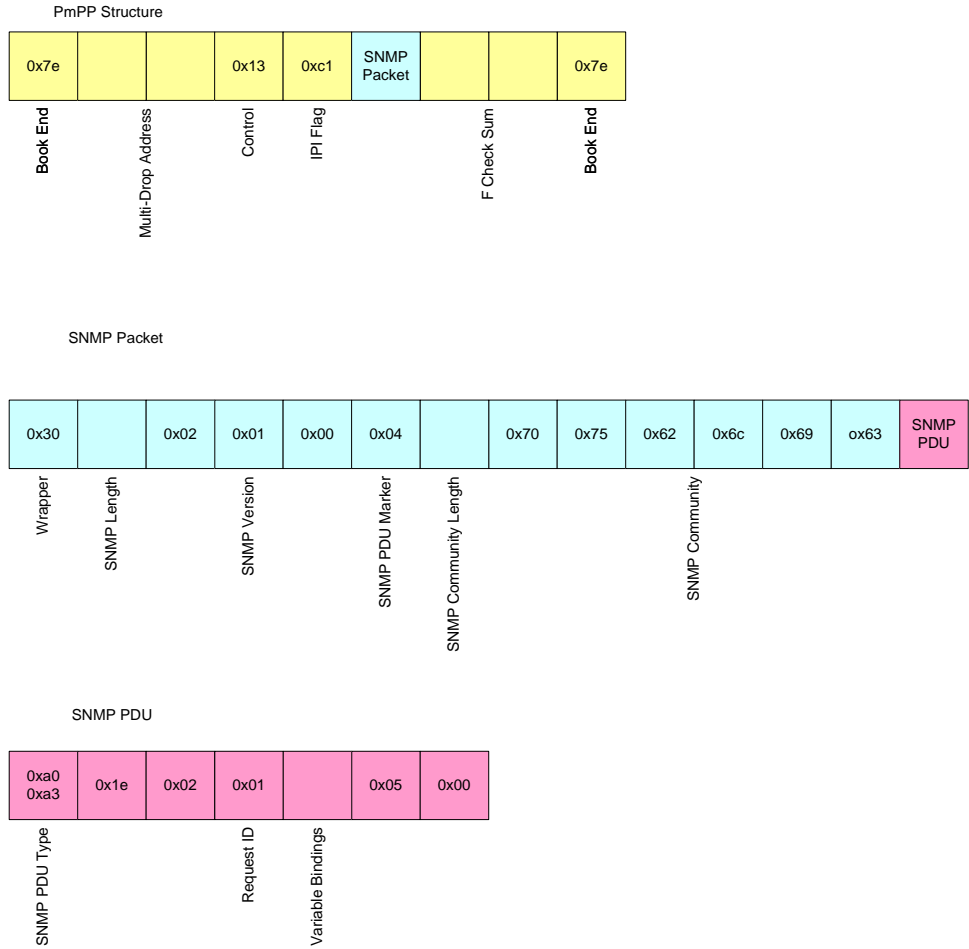


Figure 19 Protocol Packet Diagram

3.3.2 Records

The Record interface is specified in the Java Connector Architecture. It is the representation of the information to be sent and received from external devices. In the CTMS system the record concept is implemented by the NTCIPSingleRecord and NTCIPGroupRecord, and the and the NTCIPFactory.

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

3.3.2.1.1 NTCIPFactory

The NTCIPFactory internally holds a collection of MIBObjects and can return them to the caller as NTCIPSingleRecord or NTCIPGroupRecord.

3.3.2.1.2 NTCIPSingleRecord

This class implements the Record interface and specifies the properties for communication with a SNMP NTCIP compliant device. It adds the SNMP Community Name, the SNMP request id and the SNMP MIB object that contains the OID field. The class also has an encode() and a decode() method. The encode() method uses SNMPPduGet and SNMPPduSet to encode requests to the device. The device uses the SNMPPResponse to decode the SNMP packet received from the device. The end value from the SNMP process is set in the MibObject value property.

3.3.2.1.3 NTCIPGroupRecord

Like the NTCIPSingleRecord class, this class also implements the Record interface and in addition adds the ability to build a single record with a group of MIB objects. This enables them to be executed all at once on the device to reduce communication overhead.

3.3.2.2 Record Class Diagram

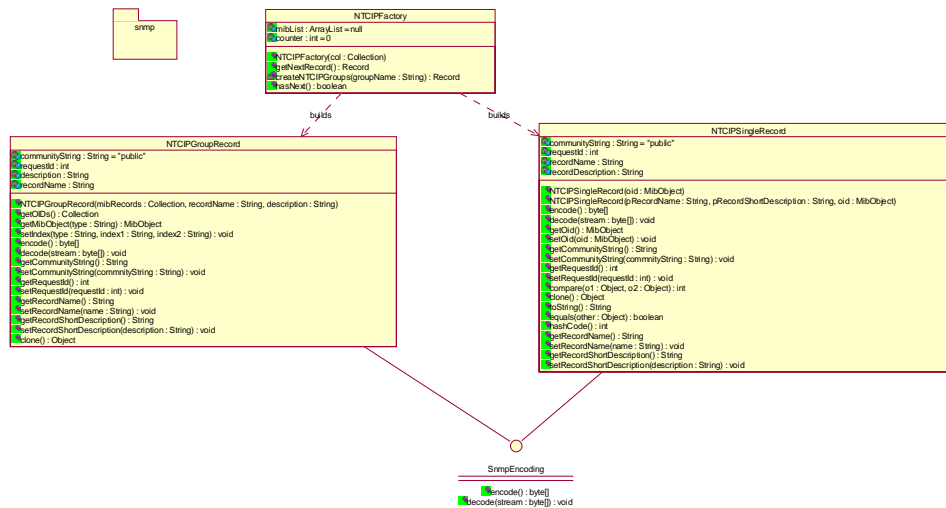


Figure 20 Record Class Diagram

3.3.3 SNMP

3.3.3.1 Purpose

The SNMP Package is responsible for encoding and decoding the SNMP sets and gets to and from the devices. There are only 3 non-abstract classes to use from this package. The CTMSSnmpPduSet, CTMSSnmpPduGet and the CTMSSnmpPduResponse. The

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

NTCIPSingleRecord and NTCIPGroupRecord classes have the only interface to use these in the system. These have an encode() and a decode() methods. The encode method uses the CTMSSnmpPduGet and CTMSSnmpPduSet to read and write data from the device. SNMPPduReponse class is used to decode the value received from the device.

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

3.3.3.2 SNMP Class Diagram

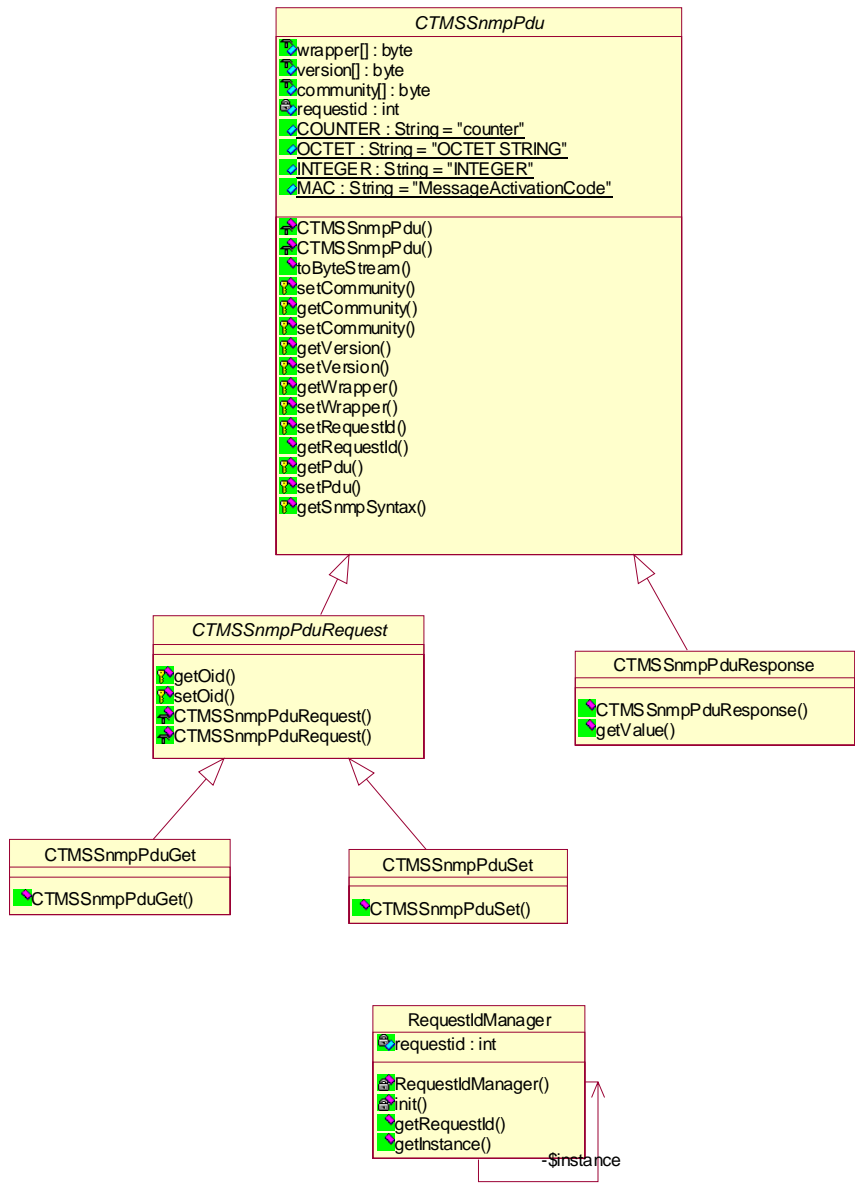


Figure 21 SNMP Class Diagram

3.3.3.2.1 SNMPPdu

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

This class represents the common elements between the request and response encoding requirements needed to create a SNMP message. The community string, wrapper, SNMP version and request id. The community string setter method encodes the String to the hex equivalent or sets the community string to the hex value of "public." This is left over from the CHART2 source code. The remaining values are set in byte array format with no further encoding. It provides one other method that translates the OID type to the correct SNMPSyntax object.

3.3.3.2.2 SNMPRequest

This adds only the attribute OID to the class and is still abstract.

3.3.3.2.3 SNMPResponse

The SNMPResponse class is the mechanism used to unpackaged the SNMP message from the device. In the constructor the class SNMPRequest from the JoeSNMP library is used with a BEREncoder to decode the value. Then the request id passed in is compared to the request id from the SNMPMessage. If they do not match an exception thrown. The method getValue() is used to get a string representation of the value in the message. The method was taken from the CHART2 source code. Is uses the SNMPSyntax and lots of other classes from the JoeSNMP library to get the value. In addition, there are additional methods that can return a byte array, an array of strings, or an array of byte arrays. The array methods are used to decode responses to groups of MIB calls.

3.3.3.2.4 SNMPPduGet

This class extends the SNMPRequest and is a concrete implementation. The Constructor finishes the last detail of the request, which is the adding the correct variable binding to the SNMP request message before its encoded. It simply uses the parent class method, and then adds the answer to the SNMPRequest property. It uses multiple variable bindings to get a group of responses at once.

3.3.3.2.5 SNMPPduSet

This class extents the SNMPRequest and is a concrete implementation. The Constructor finishes the last detail of the request, which is the adding the correct variable binding to the SNMP request message before its encoded. It simply uses the parent class method, and then adds the answer to the SNMPRequest property. It allows the variable binding to be either a string or a byte array.

3.3.3.2.6 RequestIdManager

The singular instance of the RequestIdManager. The object iterates the requestid by one until it reaches 255. Once the requestid is greater than 255 it will be set to 1. This is done in order to make sure that the responses are correctly matched to the request.

3.3.3.3 SNMP Encode Sequence Diagram

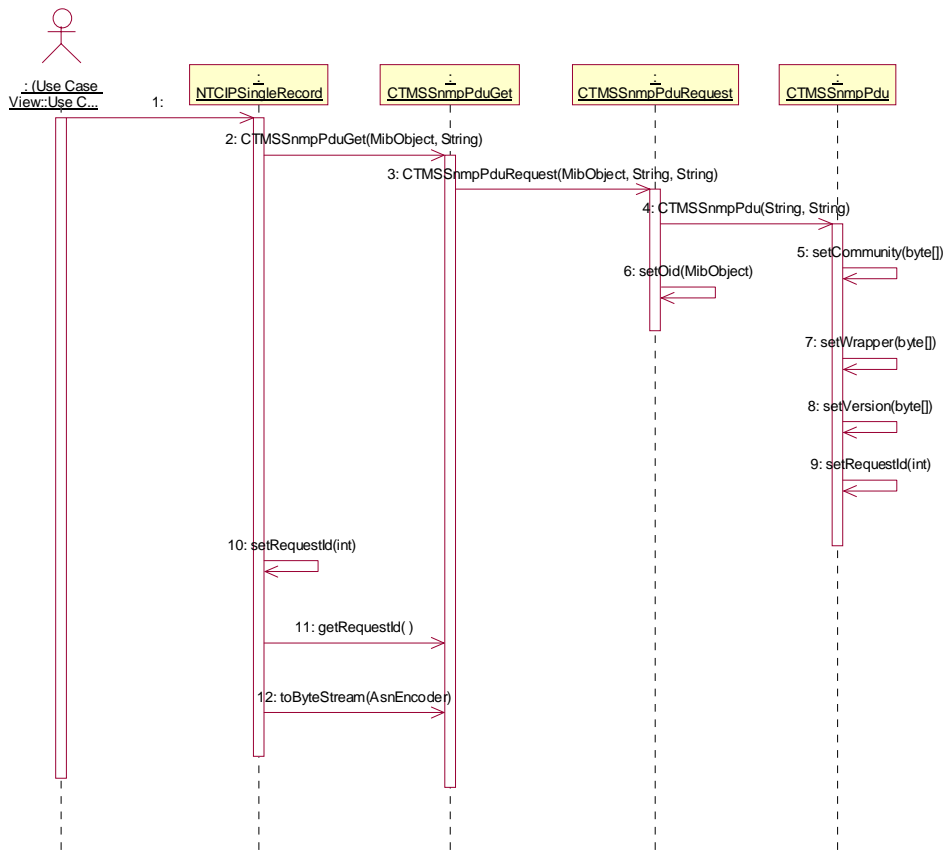


Figure 22 SNMP Encode Sequence Diagram

The process starts with the NTCIPSingleRecord(or NTCIPGroupRecord) encode() method being called. This method instantiates a CTMSSnmpPduGet object. This object calls the super constructor, CTMSSnmpPduRequest. This class calls the CTMSSnmpPdu constructor. The class set the common sttributes, community string, wrapper, version and request id. The CTMSSnmpPduRequest class then set the OID object(or multiple OIDs). The CTMSSnmpPduGet class the uses the JoeSnmpLibrary SNMPRequest class, REQUEST_TYPE flag in the constructor. The object is a property of the CTMSSnmpPdu class.

3.3.3.4 SNMP Decode Sequence Diagram

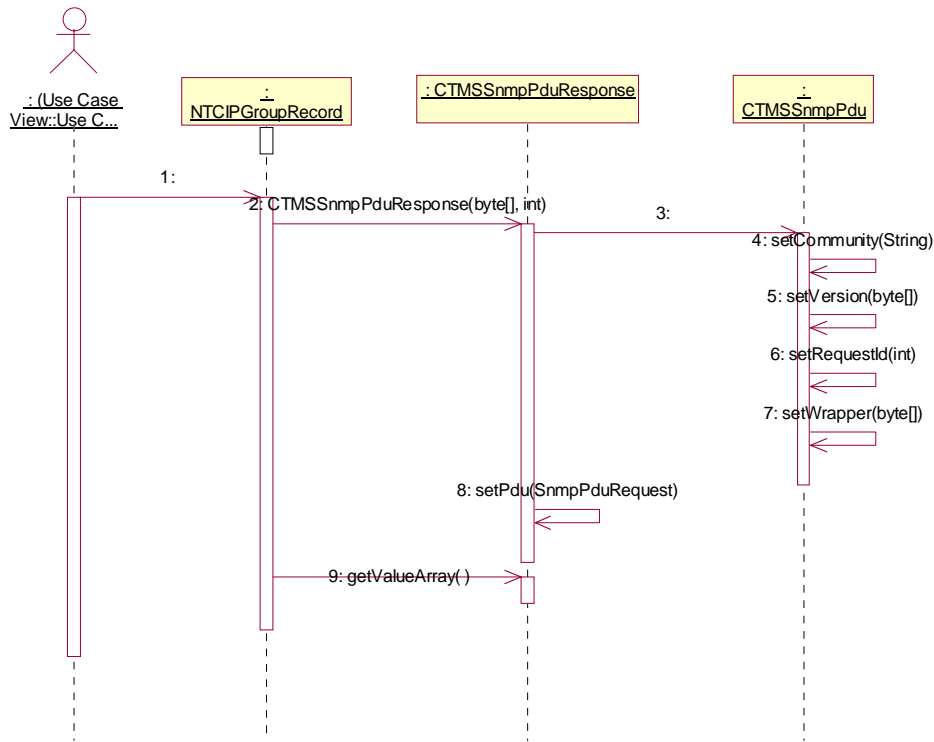


Figure 23 SNMP Decode Sequence Diagram

This sequence begins with the decode method being called on the NTCIPSingleRecord or the NTCIPGroupRecord classes. The class instantiates a new CTMSSnmpPduResponse. The response calls the parent class constructor, CTMSSnmpPdu, to set the following attributes, community, version, request id and wrapper. The CTMSSnmpPduResponse uses a SNMPRequest object from the JoeSNMPLibrary to decode the response. Once the NTCIPRecord class calls the getValue() method, the final decoding takes place and the value is returned. This getValue method is from the CHART2 source code.

3.3.4 Management Information Database

Management information bases (MIBs) are a collection of definitions, which define the properties of the managed object within the device to be managed. Every managed device keeps a database of values for each of the definitions written in the MIB.

3.3.4.1 Purpose

These classes are the mechanism that reads the management information objects from the CTMS MIB XML file. The reader reads only the mapping file “mib.xml” mapping file.

3.3.4.2 MIB Class Diagram

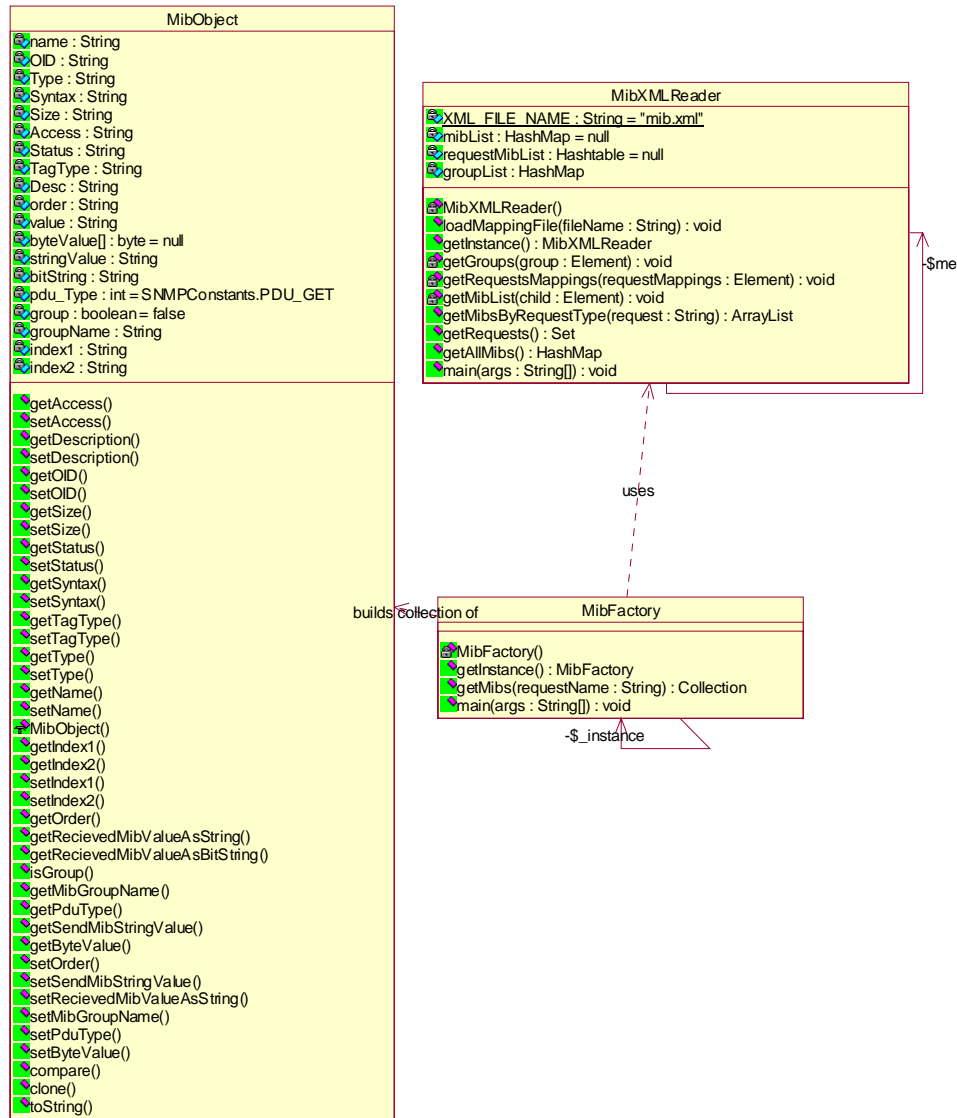


Figure 24 MIB Class Diagram

3.3.4.2.1 MibXmlReader

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

The Managed Object Information file has seven attributes associated with each object. Each element in the file started with the attribute “mib” represents 1 SNMP object. The following is the list of attributes:

- OID – The numeric values that identifies this object
- Type – The alpha designation of the object
- Syntax – The SNMP type of the object
- Size – The max size of the object
- Access – Indicates if the object has read or write permissions
- Status – Indicates if the object is required
- Tag type - Indicates if this object was specified in the NTCIP MIB or a manufacturer specific MIB.

In addition, there is the following one element:

- Description – Describes the object

3.3.4.2.2 MibObject

This class represents one object for the Managed Information Base file.

3.3.4.3 MIB Sequence Diagram

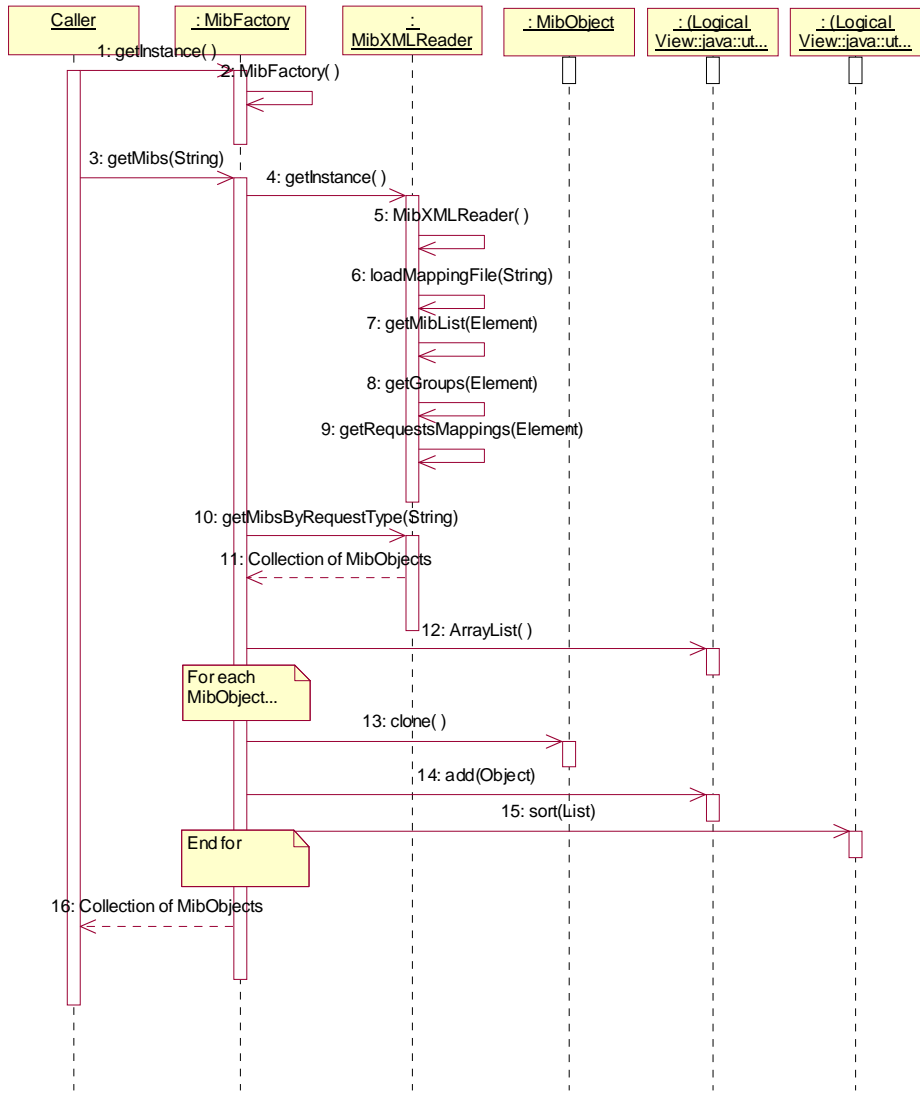


Figure 25 MIB Sequence Diagram

3.3.4.4 XML File Definition

The definitions of all the MIBs used in CTMS and the grouping of MIBs for each request and manufacturer and technology is listed in the mib.xml file.

Here is a sample definition of a few MIBS:

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

```
<mib oid="1.3.6.1.4.1.1206.3.18.2.3.9.14.0" type="surgeCommStatus" syntax="INTEGER" size="0-255"
access="read-only" status="optional" tagtype="NTCIP" >
  <description>Surge Comm Status</description>
</mib>
```

```
<mib oid="1.3.6.1.4.1.1206.4.2.3.3.4.1.3" type="characterBitmap" syntax="OCTET STRING" size="0-255"
access="read-only" status="mandatory" tagtype="NTCIP" >
  <description>Character bitmap</description>
</mib>
```

```
<mib oid="1.3.6.1.4.1.1206.4.2.3.7.6.0" type="dmsIllumManLevel" syntax="INTEGER" size="0-255"
access="read-write" status="mandatory" tagtype="NTCIP" >
  <description>Manual Brightness Value</description>
</mib>
```

The following is a sample of the MIBS that need to be executed to poll a LED based Skyline sign. It has seven single MIBS and 2 group MIBS. It shows the MIBS and the order in which they need to be executed.

```
<request type="request_poll_skyline_led">
  <include-group name="led_status" order="1"/>
  <mibName value = "dmsMessageMultiString" order="16"/>
  <mibName value = "dmsMessageOwner" order="17"/>
  <mibName value = "dmsMessageRunTimePriority" order="18"/>
  <mibName value = "feedbackNumPages" order="19"/>
  <mibName value = "feedbackRows" order="20"/>
  <mibName value = "feedbackColumns" order="21"/>
  <mibName value = "feedbackData" order="22"/>
  <include-group name="brightness-status" order="25"/>
</request>
```

The following is the definition of a group of MIBS:

```
<group name="brightness-status">
  <mibName value = "dmsIllumControl"/>
  <mibName value = "dmsIllumNumBrightLevels"/>
  <mibName value = "dmsIllumBrightLevelStatus"/>
  <mibName value = "dmsIllumLightOutputStatus"/>
</group>
```

This separation of MIB definitions and grouping of MIB usage allows us to reuse the same MIB definition across multiple requests and vendors.

3.4 Communication Services

In addition to communicating with devices, the communications layer has additional functions that allow it to be integrated with other services and components. These include providing synchronous access to data pertaining to devices, dmses, connections, connection pools and other resources through the delegate layer, handling success and failure of asynchronous instructions, etc.

3.4.1 Delegate channel classes

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

After an instruction has been executed by the Comm Server, the CTMSClientContainer posts a success or a failure event (CTMSCommEventDTO) to the CommChannel through the CommChannelSender. The event contains all the necessary information needed to process it. Once the event has been posted to the CommChannel, the Comm Server relies upon the delegate layer to process the event appropriately. It is the responsibility of the delegate channel classes to handle these event appropriately.

3.4.1.1 delegate.channel.CommChannelMDB

This consumes the messages posted to the CommChannel. When it's onMessage() method is called, it uses the CTMSCommChannelCmdFactory to create the CommChannelCmd interface to the concrete command class that will handle the event. It also sets the ActionController to used by the CommChannelCmd before executing the command.

3.4.1.2 delegate.channel.comm.CTMSCommChannelContext

This class implements the CommChannelContext interface and contains the information needed to process a command like the instruction id, owner, type, device id etc.

3.4.1.3 delegate.channel.comm.CTMSCommChannelFailedContext

This extends CTMSCommChannelContext to include the cause of the failure.

3.4.1.4 delegate.channel.comm.CTMSCommChannelCmdFactory

CTMS implementation of CommChannelCmdFactory to create a command object that handles the appropriate instruction reply from the Comm Server. It looks at the event name and creates either a success or a failed command. For example, if a poll succeeded, it creates PollDmsCmd. If a poll has failed, it creates PollDmsFailedCmd.

3.4.1.5 delegate.channel.comm.CommSuccessCmd

It is an abstract class that implements BaseCommChannelCmd and executes the logic common to all success commands like updating the instruction status to completed, posting the results to the operator queue and broadcasting the instruction status change. It has an abstract method executeSuccess() that allows subclasses to implement logic specific to them. If any exception is raised during the execution of the subclass specific logic, it treats it as a failure of the instruction and updates the status of the instruction to failed. It also logs the failure and posts the CTMSCommFailedDTO to the operator queue.

3.4.1.6 delegate.channel.comm.CommFailedCmd

It is an abstract class that implements BaseCommChannelCmd and executes the logic common to all failure commands like updating the instruction status to failed, posting the results to the operator queue and broadcasting the instruction status change. It has an abstract method executeFailure() that allows subclasses to implement logic specific to them.

3.4.1.7 delegate.channel.comm.CTMSCommFailedDTO

It implements the CommFailed interface and has all the information pertaining to the failure of the instruction.

3.4.1.8 Comm Channel Class Diagram

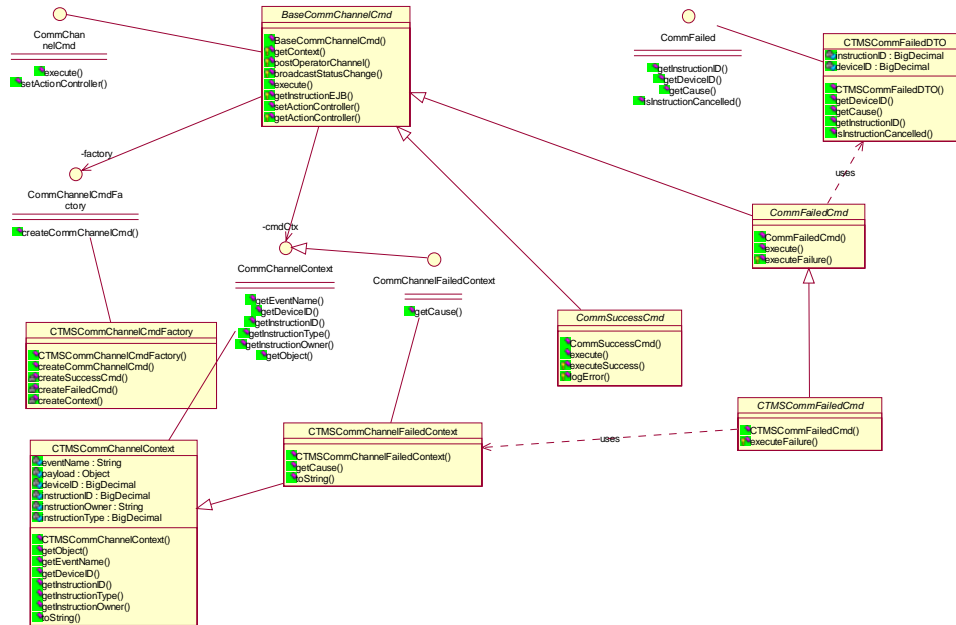


Figure 26 Comm Channel Class Diagram

3.4.1.9 delegate.channel.comm.cmd Package

The package delegate.channel.comm.cmd has all the sub classes that implement specific success and failure commands. All these classes follow a repeatable pattern.

- PollDmsCmd: This class saves the results of the poll that were sent in the CommChannelContext. It sets the statues of the poll to “success”. It relies upon the delegate action framework to save the data to the database. It then logs the success of the poll and builds a HashMap with the necessary DTOs and returns it. The CommSuccessCmd then posts it to the operator queue.
- ActivateMsgCmd: It follows the same pattern as the PollDmsCmd and saves the data to the database through the delegate action framework, logs the event and returns a HashMap with the necessary DTOs. In addition, if the reminder time was set during the activation of the message, it schedules a job to fire at the time specified. The job might send an email or an alarm depending upon the specific business need.

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

- AdjustBrightnessCmd: It saves the data, logs the event and returns a HashMap with the appropriate DTOs.
- ReadFromSignCmd: In addition to saving the data, logging the event, this class also has the responsibility of updating the status of the device to OK (Only if the current status is NEW).
- ClearDmsMsg: It saves the data, logs the event and returns a HashMap with the appropriate DTOs.
- PollDmsFailedCmd: This class gets the results of the last poll, updates the status of the last poll to failed, sets the device status to failed, logs the failure and returns the CTMSCommFailedDTO and the poll data in a HashMap. The CommFailedCmd then posts it to the operator queue.
- ActivateMsgFailedCmd, AdjustBrightnessFailedCmd, ReadFromSignFailedCmd and ClearDmsMsgFailedCmd follow the same pattern as the PollDmsFailedCmd.

3.4.1.10 Comm Channel Cmd Class Diagram

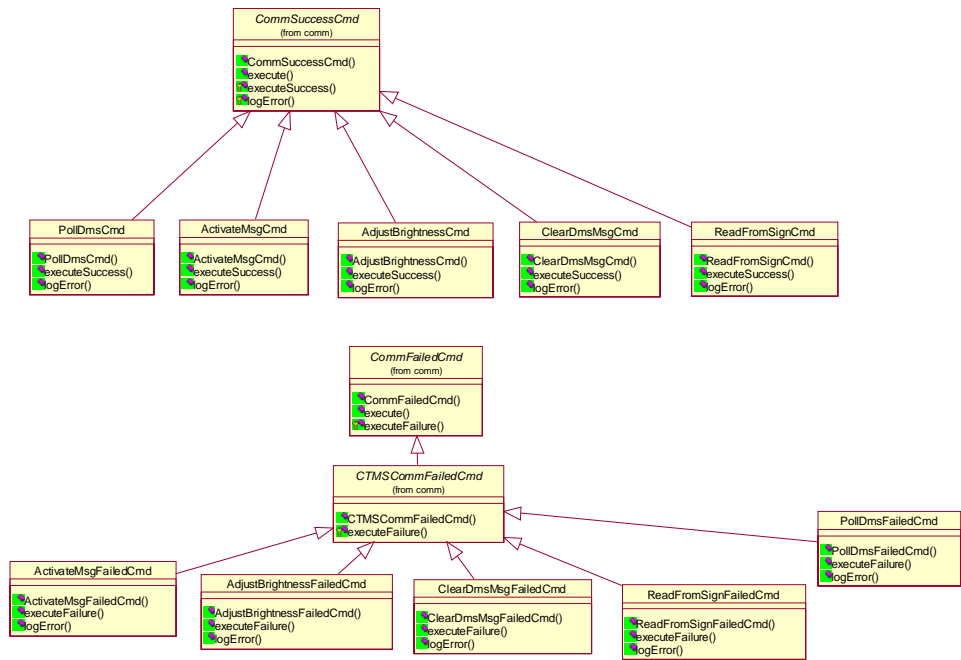


Figure 27 Comm Channel Cmd Class Diagram

3.4.1.11 Comm Success Event Sequence Diagram

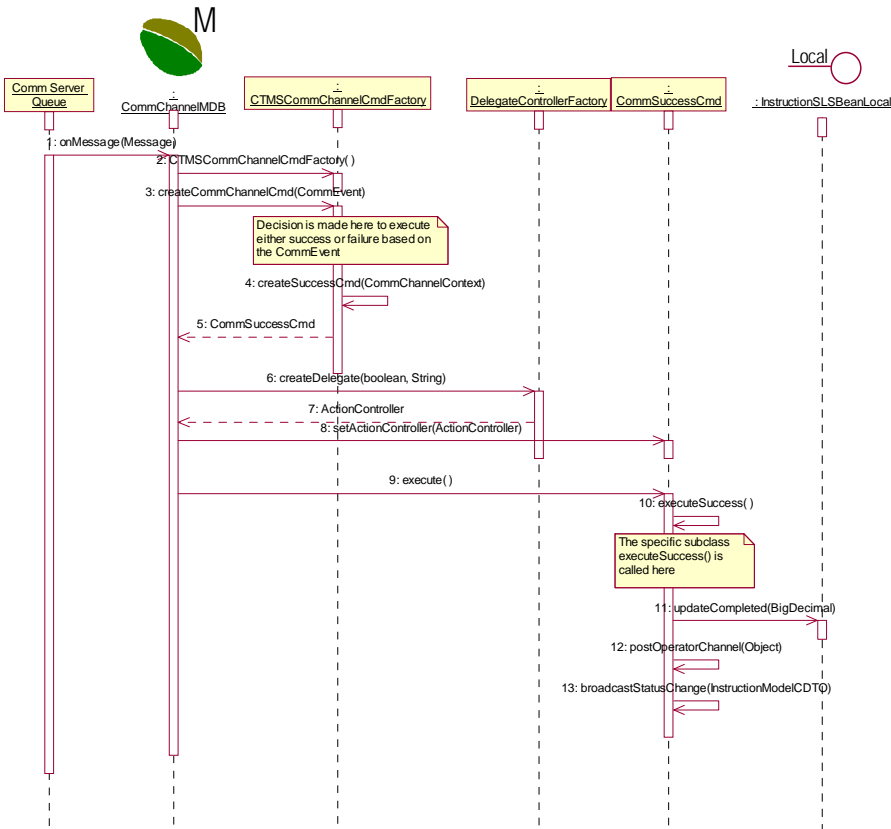


Figure 28 Comm Success Event Sequence Diagram

3.4.1.12 Comm Failure Event Sequence Diagram

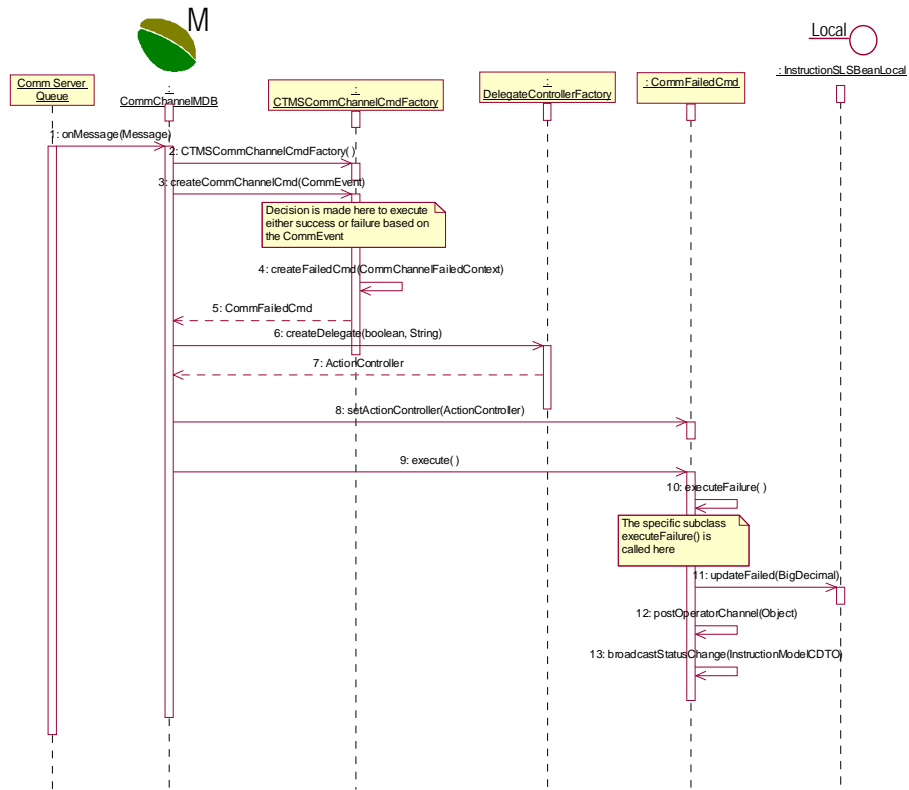


Figure 29 Comm Failure Event Sequence Diagram

3.4.2 EJBs

The Comm Server relies on a number of entity and stateless session beans to abstract the access to the database. The stateless session beans act as the facade layer through which all Comm Server database accesses are channeled.

3.4.2.1 Comm Data Facade Class Diagram

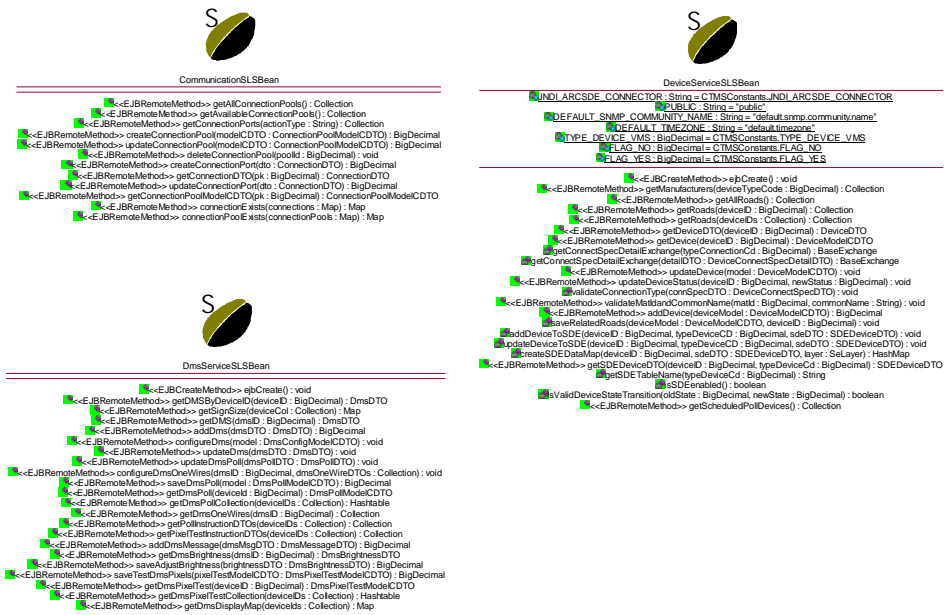


Figure 30: Comm Data Façade Class Diagram

3.4.2.2 Cdot.ctms.layer.services.comm.facade.CommunicationSLSBean

This bean contains all the façade methods needed to create and access objects related to communications like connections, and connection pools.

3.4.2.3 Cdot.ctms.layer.services.comm.facade.DeviceServiceSLSBean

This bean contains all the façade methods needed to create and access objects that are common to all types of devices. Examples of devices are DMS, RTMS, weather stations etc. The decision to keep all the common device functionality from the device specific functionality allows us to add more types of devices without touching the common functions. For example, the current iteration of CTMS only supports DMSes. But in the future we can very easily add the other types of devices like RTMS and weather stations. It also abstracts access to the underlying GIS implementations.

3.4.2.4 Cdot.ctms.layer.services.comm.facade.DmsServiceSLSBean

This bean contains all the façade methods needed to create and access objects that are specific to DMSes. Some of the functions of this bean are providing configuration info for existing dmses, adding new dmses, saving and retrieving results of communication requests like poll, activate etc.

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

3.4.2.5 Cdot.ctms.layer.services.comm.data.device package

This package contains all the entity beans, DTOs and exchanges related to devices like the device type, manufacturers etc.

3.4.2.6 Cdot.ctms.layer.services.comm.data.dms package

This package contains all the entity beans, DTOs and exchanges related to dmses, their properties (like doors, brightness levels, etc), their statuses (like the number of pixel errors, lamp failures, etc) and the results of operations on them (like poll, activate message, etc).

3.4.2.7 Cdot.ctms.layer.services.comm.data.fieldcomm package

This package contains all the entity beans, DTOs and exchanges related to connection information like the connections, connection pools, device connection specifications etc.

3.4.3 Composite DTOs

The Comm Server relies on Composite DTOs to hide the actual database schema of the underlying member DTOs. This safeguards the clients of the Comm Server (like the UI) from changes to the underlying database table schemas. Some of the major ones are described below :

3.4.3.1 Cdot.ctms.layer.services.comm.data.DeviceModelCDTO

This class is a composite DTO that represents the application model of device and its related data like the connect spec, related roads, etc.

3.4.3.2 Cdot.ctms.layer.services.comm.data.DmsPollModelCDTO

This is a composite DTO that represents the results of a poll and its related data like the brightness data, doors, power supply status, controller errors, sensor errors, surge protector errors, etc.

3.4.4 Delegate Actions

All access to the Comm Server services is controlled through the delegate action framework. The following class diagram illustrates the actions used by the Comm Server :

3.4.4.1 Delegate Actions Class Diagram

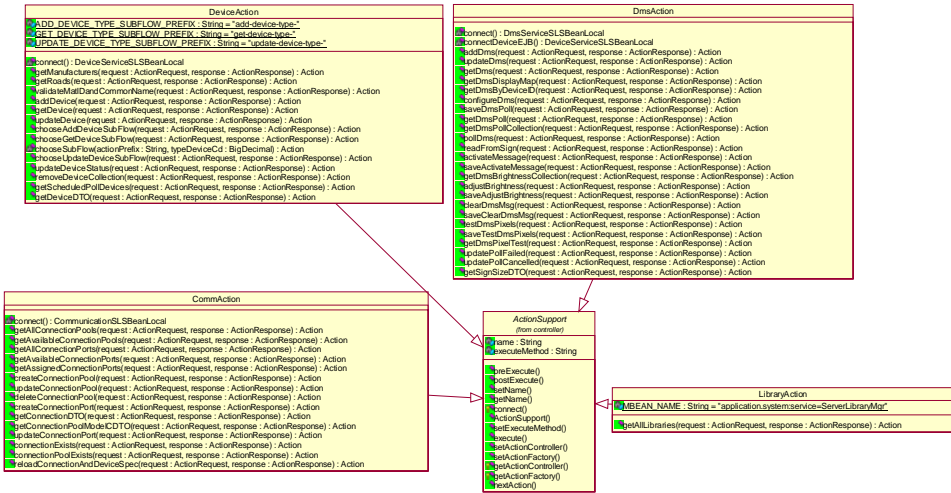


Figure 31: Delegate Actions Class Diagram

3.4.4.2 Cdot.ctms.layer.delegate.actions.CommAction

This class uses the CommunicationSLSBean to implement the logic of the actual action. The following sequence diagrams illustrates the steps needed to create a new connection pool in the createConnectionPool() method. It is a repeatable pattern used by all the other methods in the class.

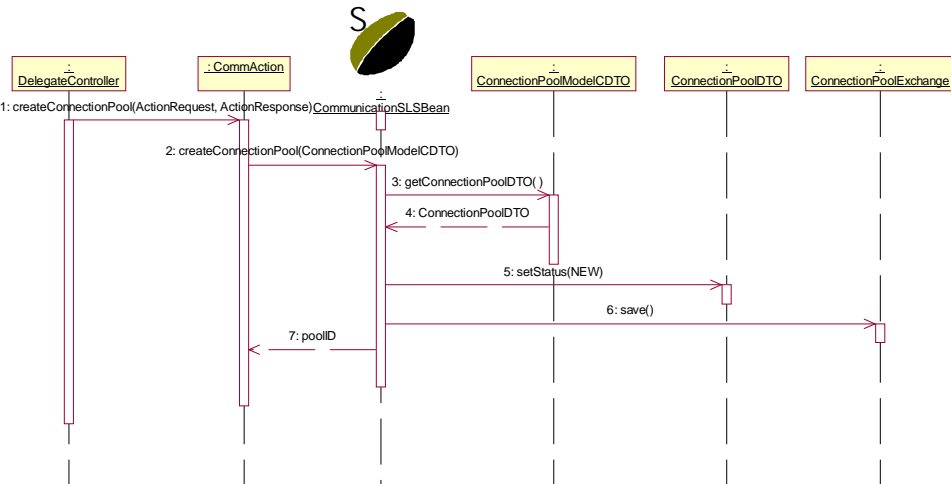


Figure 32: Add Connection Pool Sequence Diagram

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

This class is primarily used by the UI to create and update connections and connection pools, verify uniqueness of pool and connection names, etc. It is also used to reload the device connection specs whenever a new pool/connection is created or updated.

3.4.4.3 Cdot.ctms.layer.delegate.actions.DmsAction

This class is used by the UI to save and retrieve the results of dms related instructions like poll, activate message, configure dms, etc. In addition, it also provides optimized view of the data to boost performance. For example, if multiple dmses are selected to be viewed, the data required to display all the dmses is a subset of the data needed to view the complete details and is not needed until an individual dms is selected. So this action provides a method getDmsDisplayMap() that provides this minimum data as a HashMap keyed by the id of the dms.

3.4.4.4 Cdot.ctms.layer.delegate.actions.LibraryAction

This class is used by the UI to load all the available message libraries at startup.

3.4.4.5 Cdot.ctms.layer.delegate.actions.DeviceAction

Handles all service requests for the Device Service such as adding new devices, updating existing devices, validating existing devices, getting manufacturers of devices, roads devices are placed on, coordinates, etc.

3.5 Extending the Comm Server

The following section explains how to extend the Comm Server's behavior and feature set. The Comm Server can be extended and customized in the following ways:

1. Adding vendor-specific Interaction, Connection, ConnectionFactory and RecordFactory implementations to communicate to new types of devices
2. Creating custom implementations of the Comm Server's core interfaces
3. Adjusting properties to modify behavior of the Comm Server or its Containers
4. Add event listeners that can monitor the Comm Server

3.5.1 Adding Vendor-specific Extensions

Service Providers (SP - e.g., vendors who are integrating their devices into the Comm Server) must implement the required Comm API interfaces described in section 2 above. This allows new devices to be managed by the Container. The end result is that the Container can provide services such as pooling and locking and the Service Provider can focus on implementing the connection logic such as opening and closing a connection to their devices as well as encoding/decoding communications over the Service Provider connection.

The following Comm API interfaces are required to be implemented by the Service Provider:

3.5.2 cdot.ctms.layer.services.comm.cci.InteractionFactory

1. cdot.ctms.layer.services.comm.cci.Interaction
2. cdot.ctms.layer.services.comm.cci.RecordFactory

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

3. `cdot.ctms.layer.services.comm.spi.Connection`
4. `cdot.ctms.layer.services.comm.spi.ConnectionFactory`

3.5.2.1 Interaction

The Comm Server uses the Command pattern to allow functionality to be easily extended for new types of devices and vendors. Interactions are the implementation of the “command”. For each type of request a device supports, there should be an Interaction that implements the functionality. For instance, Acme’s device supports interactions called “configure” and “checkStatus”. The implementation of these interactions is specific to Acme, who implements two Interactions to handle each case. They are called “AcmeConfigureInteraction” and “AcmeCheckStatusInteraction”, respectively. The Container uses the AcmeInteractionFactory (see below) configured in it’s deployment descriptor to create the correct Interaction and then call `execute()` on it to carry out the task.

3.5.2.2 InteractionFactory

This class is responsible for returning concrete implementations of the Interaction interface. The implementation must return a different instance each time a new Interaction is requested by the Container; thus, it cannot cache Interaction references. This implementation is registered in the Container’s deployment descriptor. It must implement a default constructor to allow it to be instantiated by the Container.

For purposes of demonstration, we will use a company called Acme to show how each interface would be implemented. Acme makes roadside devices that gather traveler data. The devices need to be integrated into the Comm Server. Acme devices can perform only a few interactions such as “configure” and “checkStatus”

3.5.2.3 Connection

Each device can have proprietary protocols for encoding and decoding data sent to and from it. Thus, the SP integrating a type of device is responsible for implementing a Connection class that understands how to process information to its devices. The logic to encode and decode data is done in the `executeRecord()` method of the Connection.

In our example of Acme service provider, Acme’s devices are connected to via land lines over dial-up modems. Acme must implement a class that knows how to dial a phone number to connect to its devices, send and receive data over the dial-up connection and finally, disconnect from the device by “hanging up”. Acme creates a Connection implementation called “AcmeConnection” to perform these duties.

3.5.2.4 ConnectionFactory

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

An SP must implement a `ConnectionFactory` for each type of `Connection`. The `ConnectionFactory` implementation is configured in the `Container`, which uses the factory to create handles to devices. The underlying implementation must be serializable because it is stored in JNDI. The `ConnectionFactory` implementation must have a default constructor that allows it to be instantiated by the `Container` when it is deployed.

Back to our example of Acme, Acme needs to return instances of `ModemConnection` from its `ConnectionFactory`. So, it creates a class called “`AcmeModemConnectionFactory`” and implements the method “`getConnection()`” to return a valid `AcmeModemConnection` instance.

3.5.2.5 Create a Container Deployment Descriptor

To deploy a newly integrated set of device `Interactions` and `Connections`, you need to create a deployment descriptor and configure it to use your classes.

First, it’s easiest to copy an existing deployment descriptor and fill in the blanks with the new classes and default values.

Second, rename the `MBean` to uniquely identify the new vendor and technology. `Containers` are uniquely identified using a naming convention for the `JMX ObjectName`. To deploy your newly integrated devices, you need to name the `MBean` (e.g., “`name`” attribute) following this naming convention:

```
application.comm:server=Container,manufacturer=[Service Provider],technology=[Technology]
```

For example: `application.comm:server=Container,manufacturer=Acme,technology=VMS`

Third, modify the `InteractionFactory` field to point to your new factory class.

Finally, modify the `ConnectionFactoryClass` and `ConnectionFactoryName` attributes for any `Connections` types supported: `Modem`, `Serial`, and `Socket`. You do not have to enter all `ConnectionFactory` entries. If they don’t exist, `CTMSContainer` will ignore them. For instance, if Acme only supported modem-based devices, Acme would not have to create and register `ConnectionFactory` classes for `Serial` or `Socket`.

Note: `CTMSContainer` does not support JNDI sub-contexts, so the name must be something like: “`AcmeModemConnectionFactory`” rather than “`acme/ModemConnectionFactory`”. The later example has a sub-context of “`acme`” and a name of “`ModemConnectionFactory`” and it will not be bound into JNDI.

3.5.3 Adding Custom Implementations

As noted earlier, the `CTMS Comm Server` is a default implementation of a core set of interfaces that manage communications processing to devices. Any default `CTMS` class implementing the `Container` interfaces can be replaced. Since the default implementations cannot cover all

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

situations, you might want to create your own implementations to customize behavior for your needs. For instance, CTMSLockManager uses a shared database to store lock information for devices and connections. This is required to process locking across a cluster. However, if you didn't deploy your Comm Server in a cluster, then there is no need to have the overhead of the CTMSLockManager's locking SQL calls. Instead, you could implement and deploy your own LockManager that uses in-memory locking for better performance.

Likewise, CTMSContainerManager currently loads device ConnectInfo objects from a database table and instantiates various subclasses of CTMSConnectionSpec. However, you might need to create a custom ContainerManager that uses a different data store. To do this, you'd have to create a custom ContainerManager and plug-in the implementation. Similarly, you might need a few extra configuration attributes not available in CTMS' default ConnectInfo implementations. In this case, you'd create a custom implementation of the ConnectInfo interface from the Comm API as well as a custom ContainerManager that instantiates your version.

The point here is that the Comm Server design is flexible and "pluggable". It can accommodate other requirements as necessary without requiring code changes in the rest of the Comm Server.

The following table shows the interfaces and default implementations that can be customized:

Interface	Default Implementation	Deployment Descriptor(s)
ContainerManager	CTMSContainerManager	Communications-jmx-service.xml, CTMSContainerManager.xml
Container	CTMSContainer	Communications-jmx-service.xml, CTMSContainer.xml and any other Containers deployed
LockManager	CTMSLockManager	Communications-jmx-service.xml, CTMSLockManager.xml
Server	CTMSServer	Communications-jmx-service.xml, CTMSServer.xml

Figure 33: Container Interfaces Table

All interfaces and classes are located in the `cdot.ctms.layer.services.comm.container` package. Following the standard naming convention, the CTMS default implementation is prefixed with "CTMS".

To provide a custom implementation of any of these interfaces, follow these steps:

1. Implement the required interface. For instance, if you are replacing the Container, you must implement the `cdot.ctms.layer.service.comm.container.Container` interface.
2. Modify the JMX deployment descriptor to reference your class instead.
3. Ensure your class is in the classpath when the server deploys the service, so the MBean Server can load it properly.

Example MBean Deployment Descriptors:

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

```

<!-- Communications-jmx-service.xml -->

<service>

. . .

<mbean code="cdot.ctms.layer.services.comm.container.CTMSContainer"
name="application.comm:service=Container,manufacturer=SKYLINE,technology=VMS"
xmbean-dd="cdot/ctms/layer/services/comm/container/CTMSContainer.xml">
</mbean>

. . .

</service>

```

Figure 34: Example Communications-jmx-service.xml

```

<!-- CTMSContainer.xml -->

<mbean>
  <description>Generic container for managed devices</description>
  <descriptors>
    <persistence/>
    <display-name value="Communication Container MBean"/>
  </descriptors>
  <class>cdot.ctms.layer.services.comm.container.CTMSContainer</class>

  <constructor>
    <description>Creates Container</description>
    <name>cdot.ctms.layer.services.comm.container.CTMSContainer</name>

    . . .

  </constructor>

  . . .

  <depends>application.comm:service=ContainerManager</depends>
</mbean>

```

Figure 35: Example CTMSContainer.xml

3.5.4 Adjusting Global and Container Settings

The default Comm Server implementation provides various properties that can be configured to tune performance or modify the behavior of the Comm Server. Properties are classified as either

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

global or local depending on how they are applied in the Comm Server. They are discussed in more detail below.

3.5.4.1 Global Properties

Properties that apply to all communications are generally stored in a global location such as a properties file or a database table. CTMS stores them in the CTMS_PROPERTY database table and are accessible in the application code via the `cdot.util.ApplicationProperties` singleton. The database provides a single, shared resource for reading and updating the global properties across clustered Comm Servers and `ApplicationProperties` is CTMS' standard implementation for accessing these properties in the source.

Property Name	Description	Default Value

Figure 36: Global Comm Server Properties Table

3.5.4.2 Local Properties

Properties that are specifically for Comm Server service, vendor or device type are considered local properties. For instance, deployment descriptor properties apply at the Container-level, rather than globally. They allow individual vendors' Containers to be customized with properties like the maximum time to wait before retrying a connection, or the JNDI name of the vendor's `ModemConnectionFactory`.

Likewise, properties that are different across clustered Comm Servers are listed here. The `lock.manager.id` is currently the only one that is not a Container property.

Property Name	Description	Default Value
<code>ModemConnectionFactoryClass</code>	The full class name of the <code>ModemConnectionFactory</code> implementation	N/A
<code>ModemConnectionFactoryName</code>	The JNDI lookup name of the <code>ModemConnectionFactory</code> implementation	N/A
<code>SerialConnectionFactoryClass</code>	The full class name of the <code>SerialConnectionFactory</code> implementation	N/A
<code>SerialConnectionFactoryName</code>	The JNDI lookup name of the <code>SerialConnectionFactory</code> implementation	N/A
<code>SocketConnectionFactoryClass</code>	The full class name of the	N/A

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

	SocketConnectionFactory implementation	
SocketConnectionFactoryName	The JNDI lookup name of the SocketConnectionFactory implementation	N/A
Manufacturer	The manufacturer of the devices the Container can connect with.	N/A
Technology	The device technology. For example, VMS.	N/A
ModemMaxAttempts	Maximum number of times the Container will try to make a Modem Connection	2
ModemWaitTime	Time in milliseconds to wait between retries on a Modem Connection attempt	60000 ms (e.g., 1 minute)
SerialMaxAttempts	Maximum number of times the Container will try to make a Serial Connection	5
SerialWaitTime	Time in milliseconds to wait between retries on a Serial Connection attempt	5000 ms (e.g., 5 seconds)
SocketMaxAttempts	Maximum number of times the Container will try to make a Socket Connection	5
SocketWaitTime	Time in milliseconds to wait between retries on a Socket Connection attempt	5000 ms (e.g., 5 seconds)
InteractionFactoryClassName	The class name of the InteractionFactory implementation	N/A
lock.manager.id	This is the unique ID for a Comm Server's LockManager. It is a System property. NOTE: This is NOT a Container property, but is local to the Comm Server as opposed to global across ALL Comm Servers.	Recommend using scheme like: [hostname]-lock-manager

Figure 37: Local Comm Server Properties Table

3.5.5 Add Custom Comm Server Event Listeners

CTMS/CTIS	Version: 1.0
Communications Server Detailed Design	Date: April 20, 2005

The Container allows custom event listeners to be registered to receive notifications of Interaction events. Events in the Comm Server are implemented as JMX Notifications. JMX notifications offer a wide range of benefits including the ability to monitor them in management consoles, the ability to register interest in notifications with custom listeners, and the ability to receive them remotely.

Any implementation of `javax.management.NotificationListener` can register interest in the events. The notifications and their listeners must adhere to all rules associated with JMX notifications.

It is recommended that the handlers do not interrupt the regular processing of the Comm Server, but rather use the notifications for additional visibility into the inner workings of the Comm Server, for instance, to provide additional monitoring functionality.

3.5.5.1 Interaction Events

The Container broadcasts interaction events during the lifecycle of an interaction with a device. The Container notifies listeners when the interaction is successfully running and when the interaction is done whether failed or successful. These events map closely to the state model of an instruction; however, they are only states controlled by the Container. For instance, while the Container may notify listeners that the communications were successful, it does not mean the instruction is successful. Remember the instruction's end state relies on more than just the communications portion. It relies on the successful storing of instruction data afterwards as well. There are two ways to register event listeners.

To add a custom Interaction notification listener to ALL Containers:

1. Create a class that implements `javax.management.NotificationListener` interface.
2. Register the custom notification listener with the `ContainerManager.addNotificationListener()` method. When a Container is registered with the `ContainerManager`, the `ContainerManager` will ensure all notification listeners are registered to the Container as well.

To add a custom Interaction notification listener to a specific Container:

1. Create a class that implements `javax.management.NotificationListener` interface.
2. Register the custom notification listener by calling the Container's `addNotificationListener()` method directly.

Of course, the caller should use the JMX `ObjectName` to get a handle to the `ContainerManager` or specific Container.