

## **Communication Services Framework**

Supplementary Services and Distributed Systems through Tuned Messaging Interfaces



## Executive Summary

The Dialogic® component of the Communication Services Framework (CSF) is enabling software that makes it easier to develop sophisticated applications supporting Dialogic® products. The CSF provides developers with a high-level, object-oriented interface that abstracts much of the underlying native technology, thereby allowing the developer to focus on application development. The CSF also includes solutions for other common application primitives and component integrations, enabling developers to avoid *reinventing the wheel*.

This paper describes how to add distributed command and event delivery capabilities into the CSF for constructing distributed and web-based communications services. Sample code shows a simple, text-based interface that can be easily replaced with other protocols such as SOAP, .NET, or XML, because of the object-oriented implementation. The example also shows how to create remote network and voice device objects to allow enhanced services applications to be run remotely. An example implementation of the design described in this paper can be found in the CCSEExample classes of the CSF.



## Table of Contents

Introduction.....	2
Solution Architecture .....	2
Service Components.....	2
Command and Event Messaging .....	3
Messaging System.....	3
CSF Sample Implementation.....	4
Transport and Dispatcher (ACSInterface).....	4
Parsing and Command Factory (ACSInterfaceContext).....	4
Command Class (ADlgcCommand).....	4
Base Classes.....	5
State Machine .....	6
Methods.....	6
Transport/Dispatcher (CCSEExample) .....	8
Parser/Factory Object (CCSEExampleContext).....	9
Remote Object Implementation .....	9
Summary.....	11
References .....	11
For More Information.....	11

## Introduction

The Communication Services Framework (CSF) supports a distributed environment, whether the architecture follows Host/Terminal, Client/Server or Peer to Peer (P2P) conventions for assigning responsibilities. As part of the implementation of the Satellite PBX design (a central switching function that is controlled by distributed supplementary services applets), it was necessary to add the capability to distribute the invocation and execution of commands, and to monitor and respond to events originating from distributed processes. Once communications services are enabled with remote commanding capabilities, they are no longer tied to the hardware platform. The developer can place the application logic on remote systems. These commands can be executed at multiple levels of granularity. This applies not just to telephony resources, such as network and voice devices, but also to application objects. Adding these commands—and the distributed services they enable—elevates the Satellite PBX core from a media gateway into a service hosting platform.

## Solution Architecture

The overall concept of the CSF enabling software is to provide an advanced starting point for developers of communications services. The CSF encompasses the experience gained by solving service and integration issues while participating in, and in support of, many hundreds of communications projects. When a new solution or component is added to the CSF, it means that a particular integration has at least one solution that works. This enables developers to avoid a large number of potential design issues, greatly reducing their learning curve and giving them a running start at creating their own unique communications service applications.

### Service Components

The Satellite PBX solution delivers communications services such as voice mail and directory services using a service-oriented architecture implemented with message-passing transport adapters that optimize communications between components. Each component implements a part of the overall service. Components may be:

- In diverse locations
- In the same process
- In the same server

- Connected to the LAN
- Connected to the Internet

The transport adapters allow the higher-level logic of the service components to exchange commands and events with other components while being abstracted from the lower-layer details.

The current implementation requires that the service component Constructor has knowledge of the proper transport adapter to be used. Enhancements to the existing implementation could select the proper adapter based on a dynamic local routing table.

A service component can be a device, a media service, or an application. A service component has the following attributes:

- **Command queue** — The component accepts commands via a queue. These commands are then executed in the order received.
- **Known methods** — The component's capabilities are exposed via known methods (commands).
- **Subscribe/notify** — The component delivers events via the subscribe/notify design pattern.

There are many different types of service components, including:

- **Hardware device** — This component represents a specific hardware device (network interface, media, conference resource, etc.).
- **Media or signaling service** — Wraps a group of commands that make up a service that is repeated many times (that is, play a menu and get a DTMF response, display caller ID and ring phone). Analogous to a subroutine in a procedural language.
- **Supplemental services** — An application that, for example, provides routing selection based on least-cost routing rules, a VoiceXML interpreter, and a single level of voice mail menu logic.
- **Helper components** — Help other components do their job. While not directly addressable by other components, these objects assist with the delivery of commands and events. Examples include event processors, message transports, and data parsers.

### Command and Event Messaging

From a service component’s point of view, what is required to send a command or to subscribe to an event from another service component is to issue the QueueCommand() and the RegisterForNotification() methods of the EventProcessor class and AbstractObserverSubject classes, respectively. If the service component is located in the same process, the command or registration request is sent directly to the object by placing it on the central queue for that process using the QueueCommand() method. The command is then pulled off the queue by the event processor, which then executes the command on the component specified in the command object. More description of this design can be found in the paper *Introduction to the Communication Services Framework* (see the *For More Information* section).

If the service component is located outside the current process, it is queued to a proxy stub created to represent the component on the local system. It is then transported to the remote component via the messaging system.

### Messaging System

A distributed message delivery system is similar in design to Remote Method Invocation (RMI) architecture [Coulouris, Dollimore, Kindberg]. It can be separated into four main features:

- **Transport** – Implements the method of sending and receiving messages.
- **Dispatcher** – Sends the command object to the appropriate command target.
- **Parser** – Helper method for the dispatcher. Parses incoming messages and formats outgoing messages.
- **Command factory** – Helper method for the dispatcher. Takes parameters from the message and creates the command object to be queued.

In this implementation, the transport and dispatcher are combined into one class and the parser and command factory into another. It is planned that future CSF enhancements will divide each of the service components into individual classes.

Figure 1 is a diagram of the message flow for commands and events.

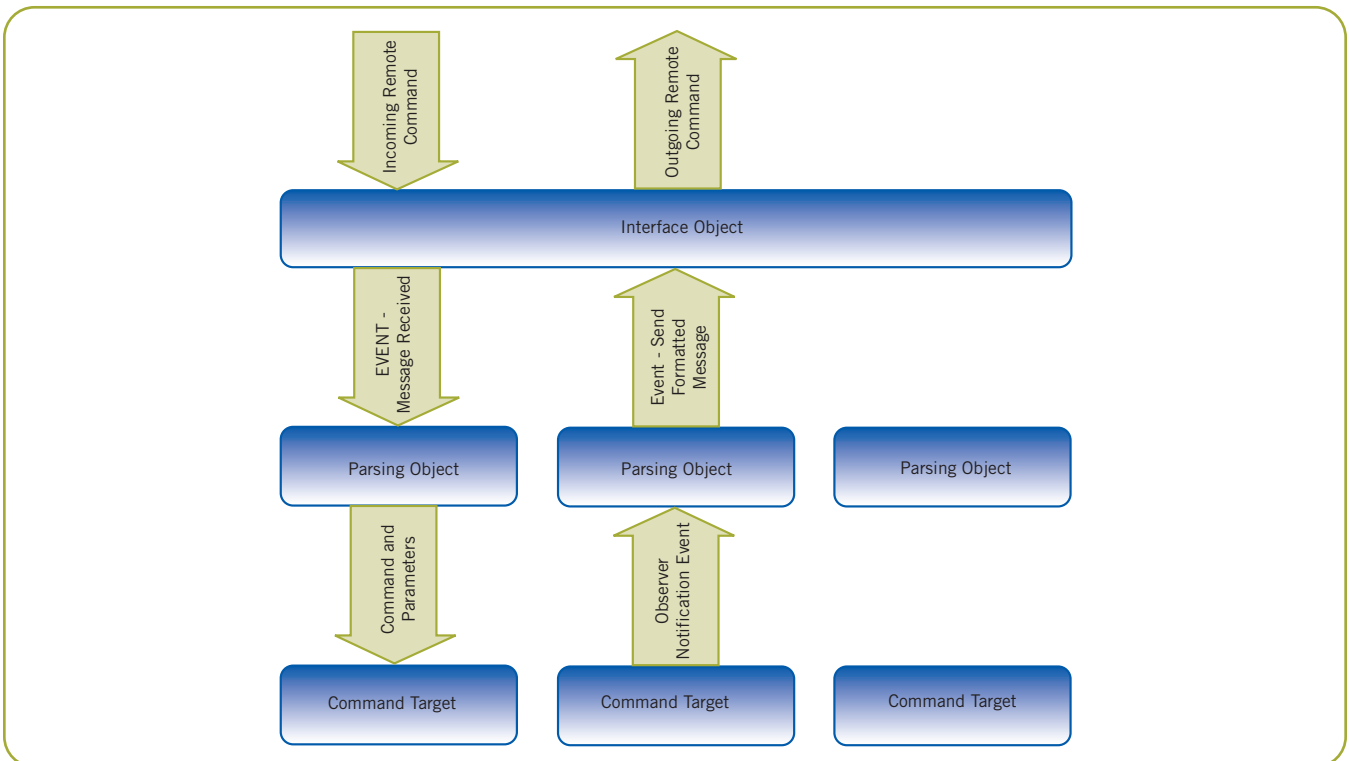


Figure 1. Remote Command and Event Message Flow

## CSF Sample Implementation

The sample implementation described in this section shows a simple, text-based interface that can be easily replaced with other protocols such as SOAP, .NET, or XML, because of the object-oriented implementation. The example also shows how to create remote network and voice device objects to allow enhanced services applications to be run remotely. For the location of the sample code, see the *For More Information* section.

### Transport and Dispatcher (ACSInterface)

The ACSInterface (abstract client/server interface) class contains the transport and dispatcher service components. The functions of this class include:

- **Separation into individual messages** — Messages may be marshaled into one transmission or separated into multiple transmissions.
- **Forwarding of messages to parsing objects** — Once an individual message is received, it is sent to the appropriate parsing object (also called an interface context).
- **Creation of new parsing objects** — When a connection to a new command target is established, ACSInterface is responsible for creating and initializing the new parser object.

During the process of making connections and receiving messages, the interface object may have to create several threads for communications with remote systems. If this is the case, the interface object will also be responsible for managing and destroying these threads.

### Parsing and Command Factory (ACSInterfaceContext)

The responsibility of the parsing object is to translate the message into a command and an associated parameter set.

For inbound-initiated connections, the parsing object is created by the interface object when a new connection is requested. For outbound-initiated connections, the parsing object is created when connecting to a remote system. When the parsing object receives an initialization event, the appropriate service component is identified and connection information is sent to the initiator.

The parser is also responsible for observing the command target for events that may need to be transferred back to the remote side. These events are translated into the appropriate messages using a map between event ID and messages embedded in the command target itself.

### Command Class (ADlgcCommand)

Command objects allow for the execution of functions without knowing the details of the operation, receiver, or sender. This is done by using the Command Pattern [Gamma, Helm, Johnson, and Vlissidies].

In its basic form, the Command Pattern consists of a base class with a single pure virtual method that executes the desired behavior. Command objects inherit from this class and implement the execute method. These commands can be queued to an event queue and their functions completed simply by calling the Execute method.

In the CFS, the following methods are implemented in the base class:

- **SetParameters** — Receives a list of strings that are the parameters for the desired function.
- **GetAvailable** — Static function that retrieves a command object of a particular type from the object pool.
- **Subject** — Target of the command to be sent. Based on the type of command (network, voice, etc.) this may be dynamically cast from a command target to the appropriate type for the command.
- **Execute** — The functionality of the command object.

Each object keeps a map of the commands it supports in a structure. When a command arrives, the command string is used as a key to retrieve a pointer to the GetAvailable() function. If no function is found, then the command is not supported. A “Command Not Supported” type of message is returned to the requesting service component; the actual message depends on the pre-existence of such a message in the transport layer protocol.

If the function is found, it is called, target component and parameters are set, and finally the command object is queued to the target’s event processor.

## Base Classes

### *CCSExample (ACSInterface)*

Figure 2 shows the interface class inheritance.

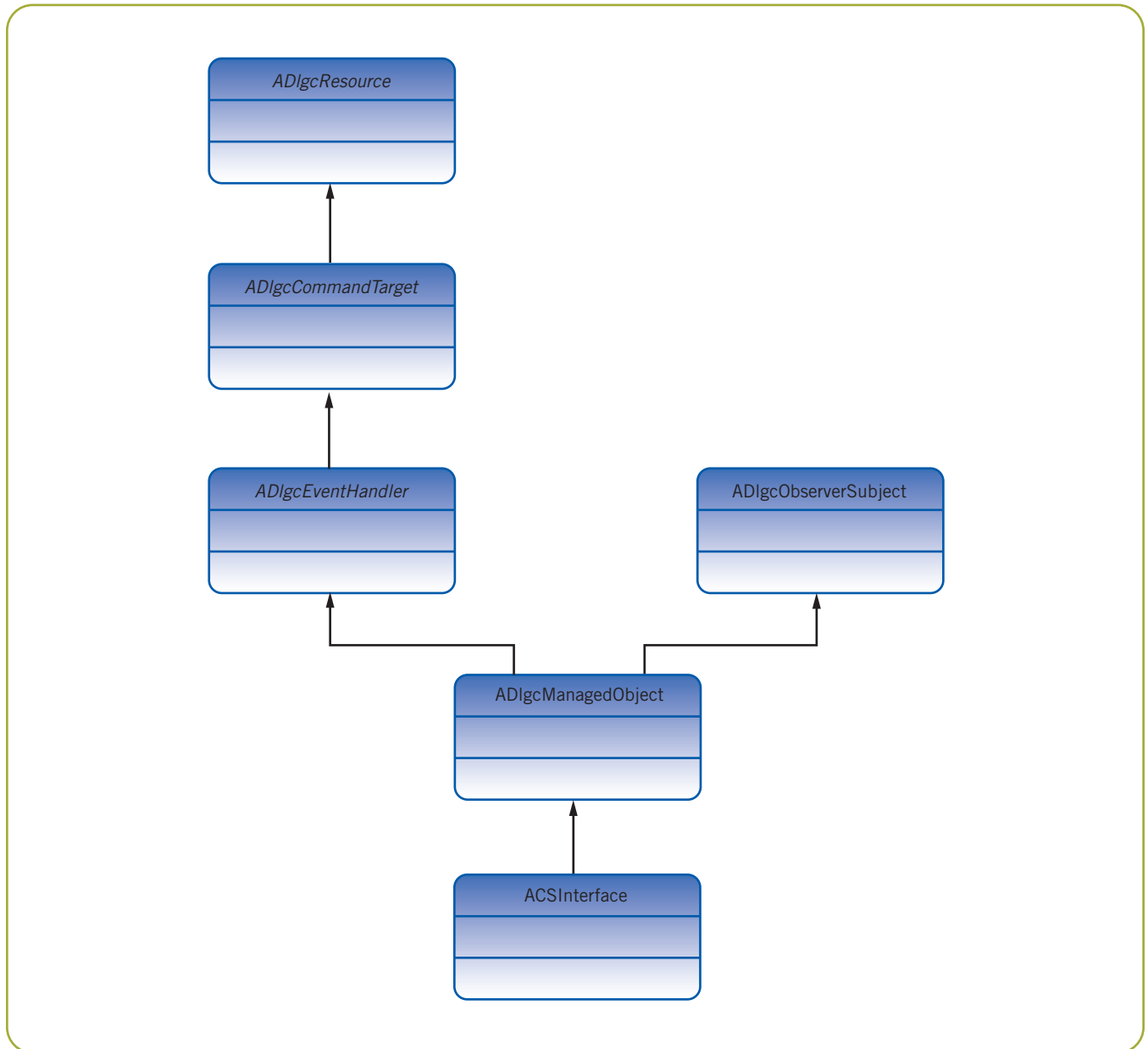


Figure 2. Interface Class Inheritance

## State Machine

The interface class inherits from the managed object hierarchy. Figure 3 shows the state machine.

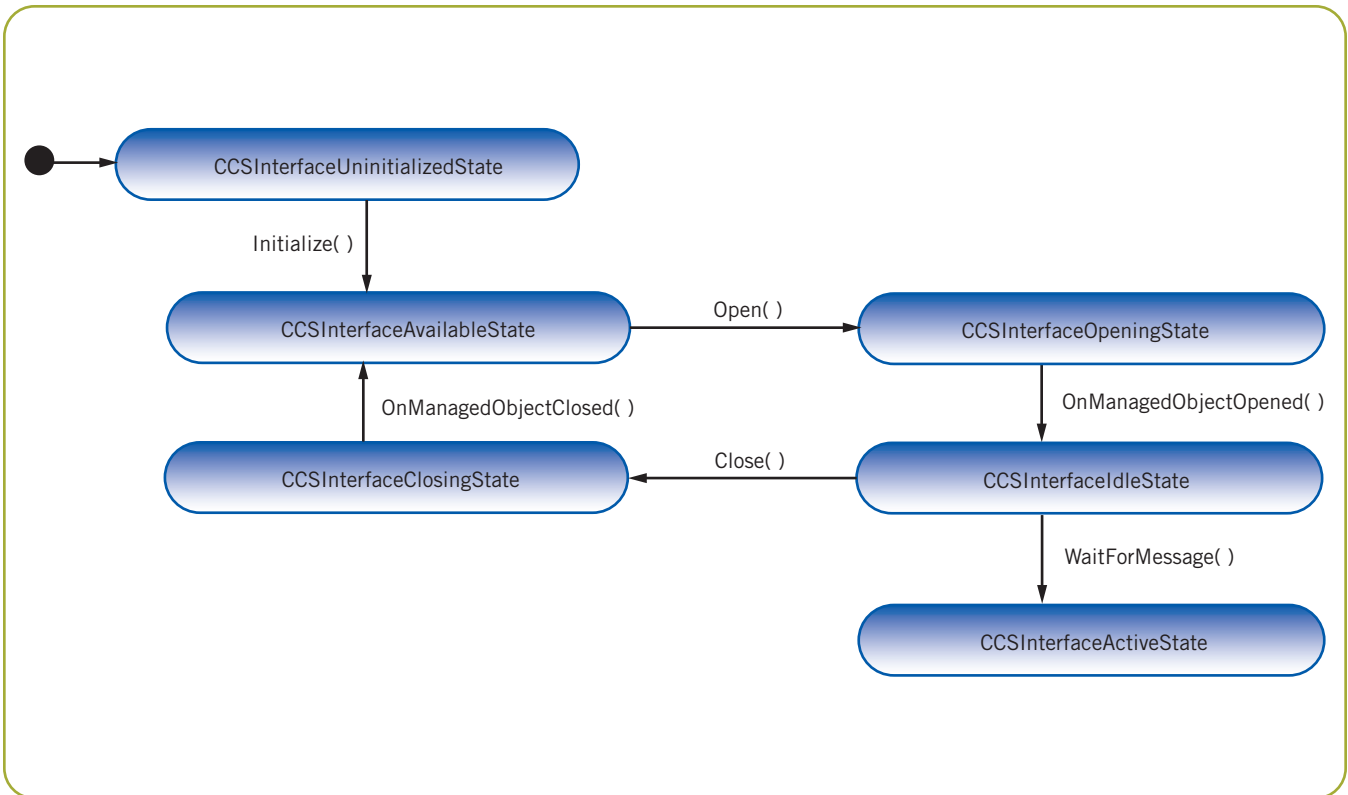


Figure 3. ACSInterface State

## Methods

Besides the standard functions required by the CSF (Handle Event, LogMessage, etc.), the interface class has the following methods:

- **int SendMessage(std::string message\_buffer, ACSInterfaceContext\* context)** — Sends message\_buffer to the remote connection indicated by context.
- **int WaitForMessages()** — Sets the interface into a mode to receive messages.
- **ACSInterfaceContext\* GetNewContext()** — Creates or retrieves a new interface context.

Figure 4 shows the parser object. The parser class has no directly called methods. The parser reacts to events delivered from an Interface object. These events cause the following primitive functions to be called:

- **PrimInitialize(CDIgcMessageEventData\*)** — Causes the parser object to obtain a service component based on the received message. If successful, an initialization complete event is generated. This initialization function is called when a request for a local service component has been received from another service component.

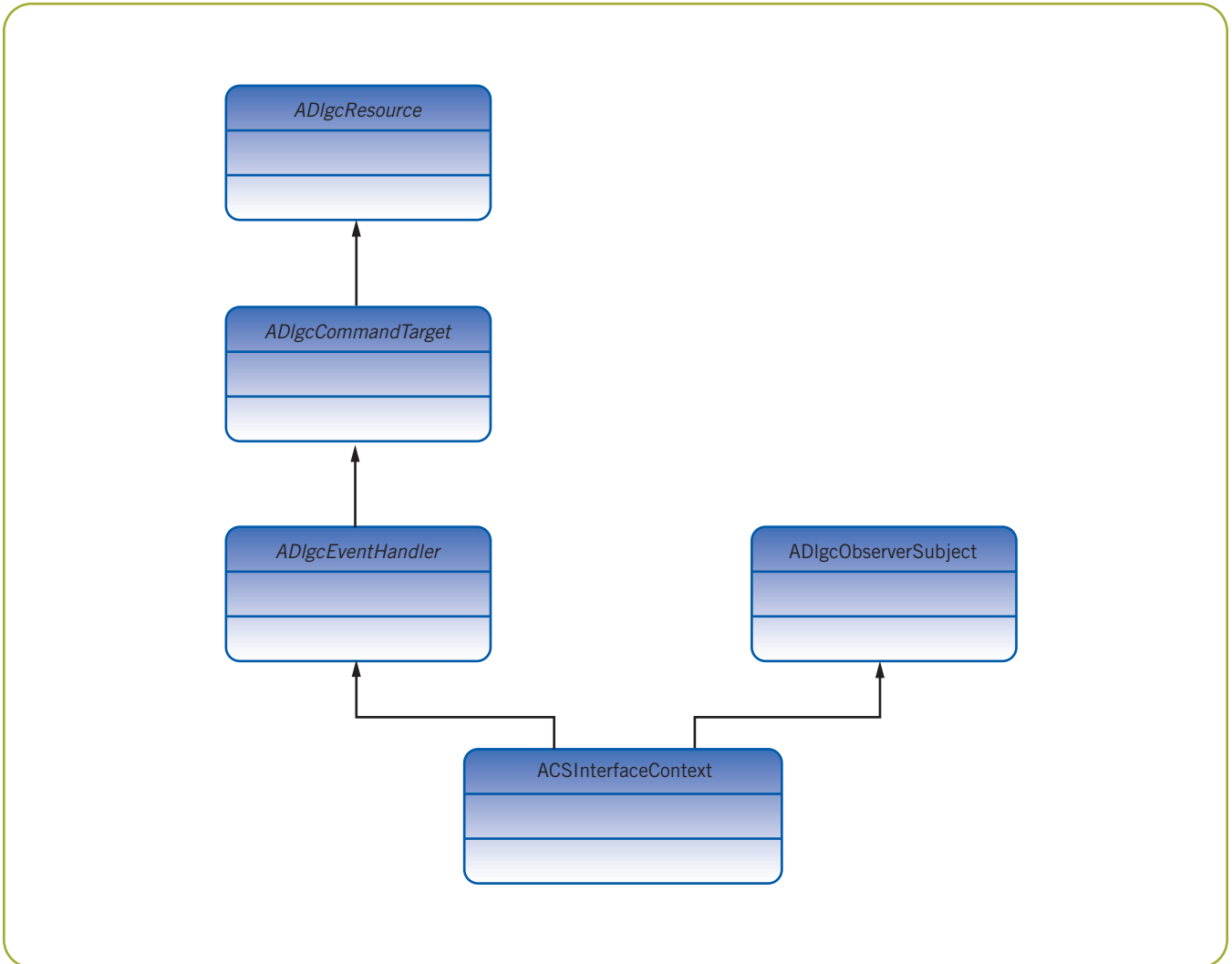


Figure 4. Parser Object (`ACSInterfaceContext`)

- **PrimInitRemote(CDIgcMessageEventData\*)** — Initializes objects that connect to a device on another server. This initialization function is used when a local service component is requesting to access another service component located in another process or system.
- **PrimInitComplete(CDIgcMessageEventData\*)** — Indicates that initialization has been completed. This function formats and sends the initialization result response to the requester.
- **PrimOnMessageProcessed(CDIgcMessageEventData\*)** — Indicates that the message has been processed. This method takes the processed message and queues the appropriate command to the target.
- **PrimOnMessageReceived(CDIgcMessageEventData\*)** — Is invoked upon receipt of a message from the interface. Parses the message buffer and queues the message processed event.
- **PrimOnServerEventReceived(CDIgcMessageEventData\*)** — Observes an event on the target device. Needs to convert the event and related data into a message to be sent.
- **PrimClose()** — Destroys the connection and causes the parser to be deleted.

**Command Target (ADlgcCommandTarget)**

Having the ability to queue commands to a command target identified by name is necessary to perform remote commanding and eventing. To do this, a map was created between the command names and functions to retrieve command objects from their respective pools.

For example:

```
_commandMap[ "PLAY" ] =
&CDlgcCmdPlay::GetAvailable;
```

The function used to obtain a command object and queue it to the target is:

```
ADlgcCommandTarget::QueueCommand
(std::string commandname, ParmList
parameters)
```

QueueCommand attempts to get a pointer to the function that can be used to retrieve the command object:

```
pooledFunction =
_commandPlay[ commandname ] ;
```

If the command is valid, a command object is retrieved, parameters are set, and the command is queued to the event processor. If the command is not valid, an error is returned:

```
commandObj = pooledFunction() ;
commandObj->Device(this) ;
commandObj->
SetParameters(parameters) ;
EventProcessor()->
QueueCommand(this, commandObj) ;
```

**Messaging Data (CDlgcMessageEventData)**

The data included with events in the remote commanding and eventing interface is encapsulated in the CDlgcMessageEventData object. The object contains:

- **std::string Command()** — Command being sent/received
- **sockaddr\_in ConnectionAddress()** — Address of remote connection
- **int ConnectionID()** — Connection ID
- **std::string Context()** — Local context ID
- **int GetNumParameters()** — Returns the number of parameters in the parameter list

- **std::string GetParameter(int ParameterNumber)** — Retrieves parameter from parameter list
- **list <std::string> GetParmList()** — Returns complete parameter list
- **std::string message\_buffer()** — Message buffer being sent/received/processed
- **std::string RemoteContext()** — Context on the remote system
- **int ServerEvent()** — Observed event on the command target

**Example Command System**

To demonstrate how to create a remote commanding system, an example implementation was developed. This implementation uses clear text to send commands and events. The transport for the sample is a standard TCP/IP socket interface.

This example was chosen, instead of other protocols, so that the reader can understand an architecture and its implementation. Deployed systems would most likely use a more efficient protocol or one of the industry-standard protocols such as SOAP or .NET.

The syntax of these messages is as follows:

```
*CONTEXT:COMMAND:PARAMETER:PARAMETER:PARAMETER:RESPONSE_CONTEXT
```

To initiate a new connection to a device, the following syntax is used:

```
*NEW:DEVICE_NAME: RESPONSE_CONTEXT
```

The response to this would be:

```
*RESPONSE_CONTEXT:NEWACK:REMOTE_CONTEXT:TIMESLOT
```

**Transport/Dispatcher (CCSEExample)**

The CCSEExample class uses a simple socket transport to move the messages between systems. Upon calling the WaitForMessages event, a thread is created to wait for connections. Once a connection is established, a thread to receive messages is started for that connection.

When a buffer is received, the buffer is scanned and broken down into individual messages. The first field of the message contains the target context for the message. This item is stripped off and the buffer sent to that context in a received message event.

If the context is “NEW,” a new context is created, the “NEW” is stripped off, and the buffer is sent to the new context with an initialization event.

For outbound connections, the parser class (CCSEExampleContext) calls the GetConnection() method to see if there is an existing connection with the desired remote system. If not, then the interface establishes one and starts a receiver thread for that connection.

### Parser/Factory Object (CCSEExampleContext)

The parser class receives events from itself and the interface class. (See the description of the ACSInterfaceContext for details.) In this implementation, the parser separates out the command and parameters and adds them to a CDlgcMessageEventData object. Following this, a message processed event is sent and the command is queued to the target.

Besides processing events associated with receiving and processing messages, the parser class is responsible for passing events from the target object to the remote application. To accomplish this, it observes the subject for supported events and marshals the data as appropriate for the event.

### Remote Object Implementation

To demonstrate the use of remote commanding and eventing, remote network interface (CRemoteNet), and remote voice (CRemoteVox) classes were created. These classes provide basic network interface and voice capability to a remote application.

#### *CRemoteNet*

The CRemoteNet class inherits from the ADlgcNet class and as such uses the ADlgcNet state machine.

The constructor (see Table 1) sets the device name, remote system address, connection interface, and event processor.

The device will be in the uninitialized state after this method.

<code>CRemoteNet(const char* deviceName</code>	Dialogic® network interface device name on the remote system (for example, dtiB1T1, iptB1T1, etc.)
<code>const char* remoteSystem</code>	Address of remote system
<code>ACSInterface* connectionInterface</code>	Interface to be used to connect to remote system
<code>CDlgcEventProcessor* eventProcessor</code>	Event processor
<code>int control_mode)</code>	Control Mode (1 = control, 0 = Monitor)

Table 1. CRemoteNet Constructor

**Initialization: PrimInitialize()**

The initialization function will create a parser context by calling the `GetNewContext()` method in the Interface object. After calling this method, the device will be in the available state.

**Opening: PrimOpen()**

To open the device, a `CSC_INITREMOTE` event is sent to the command factory service component for this object. It will in turn construct a "new" command to the desired service component. The device will now be in the opening state awaiting the acknowledgement of the connection.

**Open Complete: OnManagedObjectOpened()**

Upon receipt of the appropriate acknowledgement to the request for a new connection, the completion of the open is signaled by the sending of the `DLGC_MANAGED_OBJECT_OPENED` event.

All other primitive functions will transmit the appropriate command messages to the remote side. These commands are queued to the parser object as `CSC_SERVEREVENT` events. Commands supported in this release are:

- Wait for call
- Accept call
- Answer call
- Make call
- Drop call
- Release call
- Restart
- SCListen
- Unlisten

***CRemoteVox***

The `CRemoteVox` object inherits from the `ADlgcVox` class and as such uses the `ADlgcVox` state machine.

The constructor (see Table 2) sets the device name, remote system address, connection interface, and event processor. The device will be in the uninitialized state after this method.

<code>CRemoteVox(const char* deviceName</code>	Address of remote system
<code>const char* remoteSystem</code>	Device name on the remote system
<code>ACSInterface* connectionInterface</code>	Interface to be used to connect to remote system
<code>CDlgcEventProcessor* eventProcessor</code>	Event processor
<code>int control_mode)</code>	Control Mode (1 = control, 0 = Monitor)

Table 2. *CRemoteVox* Constructor

**Initialization: PrimInitialize()**

The initialization function creates a parser context by calling the GetNewContext() method in the Interface object. After calling this method, the device is in the available state.

**Opening: PrimOpen()**

To open the device, a CSC\_INITREMOTE event is sent to the parser object. It in turn sends a “NEW” message to the far end. The device is now in the opening state, awaiting the acknowledgement of the connection.

**Open Complete: OnManagedObjectOpened()**

Upon receiving the appropriate acknowledgement to the request for a new connection, the completion of the open is signaled by the sending of the DLGC\_MANAGED\_OBJECT\_OPENED event.

All other primitive functions transmit the appropriate command messages to the remote side. These commands are queued to the parser object as CSC\_SERVEREVENT events. Commands supported in this release are:

- Play
- Record
- SCListen
- Unlisten

## Summary

This paper has presented an implementation of a remote commanding and eventing solution using a simple socket and text protocol. More advanced applications would most likely use industry standard protocols such as SOAP, XML, and .NET.

Since the focus of this paper has been on the transport and commanding interfaces, only basic network interface and voice functions have been implemented. Although this level of granularity will function in a small LAN environment or in the same system, using these functions across multiple systems or a WAN would result in numerous messages to carry out the simplest call flows. To optimize the messaging, higher-level functions, such as play file and get digits, and service components, such as conference bridges and voice mail objects with less granular commands, would need to be implemented.

## References

[Coulouris, Dollimore, Kindberg] Tim Coulouris, George Dollimore, Jean Kindberg, *Distributed Systems Concepts and Design*, Third Edition, Addison-Wesley, Boston, MA; 1988.

[Gamma Helm, Johnson, and Vlissidies] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissidies, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Boston, MA, 1997.

## For More Information

A Zip file containing the source code for this application note can be downloaded at <http://sourceforge.net/projects/commsvcfw>

*Introduction to the Communication Services Framework (CSF)* — <http://www.dialogic.com/goto/?8989>

To learn more, visit our site on the World Wide Web at <http://www.dialogic.com>.

**Dialogic Corporation**

9800 Cavendish Blvd., 5th floor  
Montreal, Quebec  
CANADA H4M 2V9

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH PRODUCTS OF DIALOGIC CORPORATION OR ITS SUBSIDIARIES ("DIALOGIC"). NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN A SIGNED AGREEMENT BETWEEN YOU AND DIALOGIC, DIALOGIC ASSUMES NO LIABILITY WHATSOEVER, AND DIALOGIC DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF DIALOGIC PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY INTELLECTUAL PROPERTY RIGHT OF A THIRD PARTY.

Dialogic products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

Dialogic may make changes to specifications, product descriptions, and plans at any time, without notice.

Dialogic is a registered trademark of Dialogic Corporation. Dialogic's trademarks may be used publicly only with permission from Dialogic. Such permission may only be granted by Dialogic's legal department at 9800 Cavendish Blvd., 5th Floor, Montreal, Quebec, Canada H4M 2V9. Any authorized use of Dialogic's trademarks will be subject to full respect of the trademark guidelines published by Dialogic from time to time and any use of Dialogic's trademarks requires proper acknowledgement.

The names of actual companies and products mentioned herein are the trademarks of their respective owners. Dialogic encourages all users of its products to procure all necessary intellectual property licenses required to implement their concepts or applications, which licenses may vary from country to country.