

# Extending Enterprise Messaging to Mobile Applications with QAnywhere

A Technical Overview

## **INTRODUCTION**

Application-to-application messaging is a powerful and flexible technique used in many enterprises. To date, however, the challenge of building mobile messaging solutions on a variety of mobile and wireless devices has been difficult. The complexity of implementing robust, secure, and reliable messaging on lightweight devices communicating over slow, and sometimes unreliable, networks has required extensive programming.

SQL Anywhere addresses this challenge with QAnywhere to facilitate the development of mobile messaging applications.

QAnywhere provides a comprehensive store and forward messaging solution, delivering secure, assured message delivery for distributed and mobile users. Because it was designed with a small footprint, uses system resources extremely efficiently, and has low setup and administration requirements, it is ideal for lightweight or mobile devices. QAnywhere automatically deals with the challenges of slow and unreliable connections, allowing the developer to use a familiar development paradigm and to concentrate on application functionality instead of issues surrounding connectivity, communication, and security.

QAnywhere leverages extended functionality in MobiLink (the synchronization server included with SQL Anywhere) that allows it to function as a messaging server, meaning that SQL Anywhere developers can share common resources with both their data synchronization and messaging applications from the same device. This common infrastructure for data synchronization and messaging reduces administration requirements and simplifies deployment.

## USAGE SCENARIOS

### FIELD FORCE AUTOMATION

Sales reps or field service personnel travel with wireless-enabled Windows laptops and tablets running a custom enterprise application. The mobile users need to transmit data between the enterprise and the device. The device transmits data over a wireless network that experiences unreliable coverage and dropped connections.

The enterprise implements an application-to-application messaging solution with QAnywhere. The custom application on the laptop or tablet can receive data from the enterprise throughout the day, as well as transmit data back when necessary.

If the user is out of coverage when a message is sent from the mobile application, QAnywhere stores it until a connection is made at a later time. Likewise, if a connection is lost in the middle of transmitting data, QAnywhere stores the remaining portion of the transmission and sends it when a connection is reestablished. The transmission of data is determined by specifying transmission rules based on information such as the priority of a message, message size, time of day, or the type of available network.

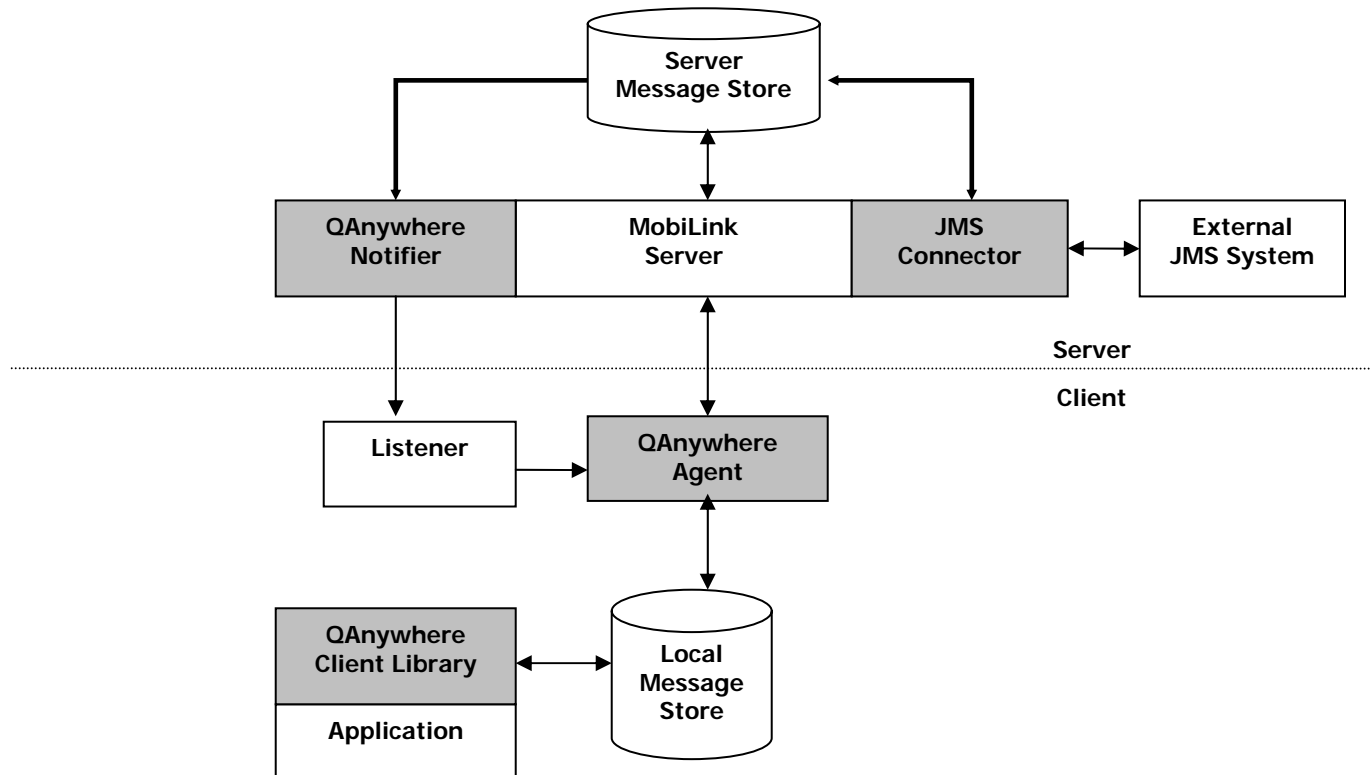
In a similar manner, if the server needs to send a message to a device, but the device is not connected, a push notification can be sent to the laptop telling it to connect to the network. This provides a way for enterprise systems to initiate communication with the mobile application.

### WAREHOUSE LOGISTICS

Warehouse employees working within a large warehouse complex each carry a handheld device that is connected over an 802.11 wireless LAN. Jobs are dispatched to the users through the handheld, and job checklists and inventory information are sent from the handheld to the back-end server.

The enterprise implements an application-to-application messaging solution with QAnywhere. The data is now transmitted back and forth between each mobile device and the server, with either of them initiating communication. As users roam between warehouses they may be assigned different IP addresses, but automatic device tracking ensures that they can always be sent messages at their current address.

## SOLUTION ARCHITECTURE



## MESSAGING ARCHITECTURE COMPONENTS

The boxes in gray in the diagram identify new components that have been introduced to implement messaging functionality by interacting with a traditional SQL Anywhere database synchronization architecture.

### QANYWHERE CLIENT LIBRARY

The client library implements the messaging API and moves messages to and from the client-side message store. The application uses this API to perform actions such as creating messages, setting their properties, and placing them on a queue for delivery, or retrieving messages from a queue. The client-side message store is a SQL Anywhere database and the server-side message store can be SQL Anywhere, Sybase ASE, Oracle, IBM DB2, or Microsoft SQL Server.

### QANYWHERE AGENT

The agent moves messages between the client-side message store and the server-side message store using the MobiLink server. Based on transmission rules, it also determines when messages should be transmitted based on a schedule, transmission rules, or a push notification. The QAnywhere Agent can remain running at all times so that message transmission can occur even when the application is not running. The Agent can also start the database to be used as the local message store and is responsible for handling all the interaction with the listener and the MobiLink server.

## QANYWHERE NOTIFIER

The notifier generates push notifications to alert the QAnywhere Agent that messages are waiting on the server for delivery to the client. It leverages device tracking functionality in MobiLink to track devices, even if new IP addresses are assigned as users roam between networks. A persistent connection can be used and push notifications are made over TCP/IP. This mode is useful when the client device does not have a public IP address, or when outbound UDP messages are not possible due to firewall restrictions. Alternatively, notifications can be sent out using UDP or SMS, depending on the device's wireless modem. When a notification is received, the agent initiates a transmission of messages.

## JMS CONNECTOR

The connector moves messages between the server-side message store and an external JMS system (such as TIBCO®, IBM MQSeries®, or JBoss®). Connector properties specify how to connect to the JMS system, which queues are used, and whether or not to compress messages. The Connector also maps message properties to their JMS equivalent headers or properties. The Connector facilitates seamless integration with the JMS system by effectively acting as a JMS client to the JMS system.

## FEATURE SUMMARY

### COMPREHENSIVE MESSAGING API

The QAnywhere API delivers a simple and powerful messaging programming interface. It provides a standard messaging paradigm that will be easy to use for developers familiar with JMS, IBM MQSeries®, or TIBCO®. Creating messages, setting message properties, putting messages on queues, and retrieving them from queues are all done easily with the API. It also allows for creating message selectors to selectively browse or receive messages from a queue, cancelling messages, and querying message status. Support for multiple queues on client devices permits multiple client applications to co-exist on a single device. Transactional messaging provides the ability to group messages in a way that guarantees that all messages in the group are delivered, or none are. Push notifications and network status changes are sent to QAnywhere applications as system messages. The API is currently available for .NET (C#, C++, and VB), C++, Java, and SQL on Windows platforms and is available for the .NET Compact Framework, C++, Java, and SQL on Pocket PC.

The basic elements of a QAnywhere message are:

- **Address** The address of this message, identified by the message store ID and the queue name.
- **Expiration** Indicates the time after which the message may expire and be removed from the message system if it has not been received.
- **InReplyToID** The message ID of the message to which this message is a reply.
- **MessageID** The globally unique message ID of the message. QAnywhere assigns this value once a message is put on a queue.
- **Priority** The priority of the message (ranging from 0 to 9).
- **Redelivered** Indicates whether the message has been previously received, but not acknowledged.
- **ReplyToAddress** The replyTo address of this message.
- **Timestamp** The message timestamp.
- **Content** The text or binary file of the message.

In addition to these message components, the developer can also create user-defined properties. QAnywhere supports text or binary messages.

To send a message to another QAnywhere client, the address is composed of a "message store-id\queue" combination. The message store-id is typically associated with a mobile device running a QAnywhere agent and the queue-name is the name of the particular queue to which the message is to be sent. Applications at this remote location can then listen for messages on this set of queues. Destination aliases can also be defined to represent a set of addresses.

## TRANSMISSION RULES

By setting transmission rules, messaging applications can send and receive data in ways that optimize performance, cost, and bandwidth. For example, high priority messages might be sent immediately over a wide area wireless network, but the transmission of very large or low priority messages could be delayed until a high-speed network is available.

Because QAnywhere supports asynchronous store-and forward messaging, it is important to distinguish between message transmission and a send/receive operation. A send operation places a message on a local queue, while a receive operation removes a message from a local queue. Neither of these operations causes a message to be moved to or from the server. An application, via the QAnywhere API, moves messages to and from local queues. A message transmission is the exchange of messages between the local queue on the client and the server. It is the QAnywhere Agent that causes a message transmission to occur. The Agent is a separate process that monitors local queues and determines when message transmission should occur. A policy tells the Agent how to determine when a message a transmission should occur. There are three pre-defined policies: automatic (the default), scheduled, and on-demand. It is also possible to define a custom policy. A custom policy is a set of rules that determine when a message transmission is to occur and which messages are to be transmitted.

### Automatic Policy

The automatic policy attempts to keep the local and server-side messages stores as up-to-date as possible. A message transmission will be performed when any of the following occurs:

- PutMessage() is called.
- A message status change has occurred. This typically occurs when a message is acknowledged by the application.
- A Push Notification is received.
- A Network Status Change Notification is received.
- TriggerSendReceive() is called.

### Scheduled Policy

The scheduled policy instructs the Agent to perform a transmission at a specified time interval, the default being 15 minutes. Given that the time interval is n seconds, the following describes how the Agent determines whether a message transmission is to occur. Every n seconds, the Agent will perform a message transmission if any of the following conditions are met:

- New messages have been placed on a local queue since the previous time interval elapsed.
- A message status change has occurred since the previous time interval elapsed. This typically occurs when a message is acknowledged by the application.
- A Push Notification has been received since the previous time interval elapsed.
- A Network Status Change Notification has been received since the previous time interval elapsed.

- Calling the TriggerSendReceive() method can be used to override the time interval. It forces a message transmission to occur before the time interval elapses.

#### On-Demand Policy

The on-demand policy will cause a message transmission to occur only when instructed to do so by an application. An application forces a message transmission to occur by calling the TriggerSendReceive() method. When the agent receives a Push Notification or a Network Status Change Notification, a corresponding message is sent to the "system" queue. This allows an application to detect these events and force a message transmission by calling the TriggerSendReceive() method.

#### Custom Policy

A custom policy allows developers to define when a message transmission is to occur and which messages are sent in the message transmission. The custom policy is defined by a set of rules stored in a file. Each rule is of the form "<schedule> = <condition>". <schedule> defines when <condition> is evaluated. All messages satisfying <condition> are transmitted. In particular, if <schedule> is automatic, the condition is evaluated when any of the following occurs:

- PutMessage() is called.
- A message status change has occurred. This typically occurs when a message is acknowledged by the application.
- A Push Notification is received.
- A Network Status Change Notification is received.
- TriggerSendReceive() is called.

#### RELIABLE AND EFFICIENT MESSAGE DELIVERY

QAnywhere leverages the inherent capabilities of the SQL Anywhere server and MobiLink to ensure guaranteed delivery of messages. QAnywhere messages can be transported over TCP/IP, HTTP, or HTTPS protocols, or from a Windows Pocket PC handheld device by ActiveSync. The message itself is independent of network protocol, and can be received by an application that communicates over a different network.

QAnywhere handles the challenges of wireless networks, such as slow speed, spotty geographic coverage, and dropped network connections. Configurable thresholds allow the developer to send messages in small groups when transmitting over unreliable networks. Guaranteed, once-and-only once delivery of messages ensures that critical data is not lost and is not delivered multiple times. The data stream is also automatically compressed for faster transmission. A "persistent connection" can be used so multiple subsequent message transmissions can be performed using the same connection, reducing the amount of setup/tear-down time required when initiating a new connection.

Automatic failover is supported, allowing a configuration where a primary and secondary server are used and the QAnywhere client will switch to a secondary server to send and receive messages if the primary server is not available.

A web redirector may also be used to deliver messages in a firewall configuration where Web traffic is allowed, but where no additional holes in the firewall are permitted.

Data stored by the QAnywhere API and data transmitted between a device and the server can also be automatically compressed using the LZ77 (deflation variant) algorithm for data transmission and storage. This can dramatically reduce network connection costs and time.

When receiving a very large message, the message is automatically broken into pieces as it is being stored in the message store to avoid consuming excessive device memory.

## SECURE MESSAGE STORAGE AND TRANSMISSION

Security of messages is preserved again by leveraging SQL Anywhere and MobiLink capabilities. Messages that are stored by the QAnywhere API and data that is transmitted between the device and the server can be encrypted using the AES algorithm and a 128-bit encryption key.

Users can be authenticated at the server using a userid/password paradigm or by using an existing authentication service provided by another application in an organization.

## PUSH NOTIFICATION

The QAnywhere Notifier is a specially configured instance of the notifier used by MobiLink server-initiated synchronization. It is configured to send notifications when a message is ready for delivery. The Listener, running on the client, receives notifications and passes them on to the QAnywhere Agent. Depending on transmission rules, a connection can then be established to receive and send messages. If a UDP notification is not possible, an SMS notification can be sent to command a device to connect to the network.

QAnywhere also supports seamless network roaming using device tracking. If a mobile device is assigned a new IP address as it moves between networks, the new address is automatically stored at the server so the user can be notified of available messages at their most recent address.

## CONNECTORS TO ENTERPRISE SYSTEMS

Seamless connection to any JMS system provides an easy point of integration to enterprise systems. JMS Connector properties are used by the MobiLink server to specify the connection parameters, queues to be used, and features such as compression of messages to be sent and received. To send a message to an external messaging system, you must specify the queue name defined by the `ianywhere.connector.address` property together with the application queue name. For example, you would specify `message store-id\jmsapp.incoming` as the destination queue, where `message store-id` is the value of the `ianywhere.connector.address` property and `jmsapp.incoming` is the destination JMS queue name in the JMS system.

When a message is delivered to a JMS messaging system, the properties are mapped into their JMS equivalents and vice-versa (a message coming from JMS has its properties mapped to its QAnywhere equivalents).

## MOBILE WEB SERVICES

QAnywhere provides support for mobile-optimized asynchronous web services. This feature leverages QAnywhere's comprehensive store and forward messaging architecture that acts as the backbone for all requests and responses. This allows mobile applications to make web service requests (even when they are off-line) and have those requests queued for transmission later. The requests are delivered as messages using QAnywhere, and then a web services connector on the server side makes the request, gets the response from the web service, and returns the response to the client as a message. QAnywhere transmission rules can control which requests and responses are transmitted based on a wide variety of parameters (network being used, size of request/response, location, time of day, and so on) The result is a sophisticated and flexible architecture that allows mobile applications to tap into the vast functionality of web services using proven technology and a simple programming model.

From a development point of view, you can work with web services proxy classes much as you would in a connected environment, and QAnywhere handles all of the transmission, authentication, serialization, and so on. A WSDL compiler is provided to take a WSDL document and generate special proxy classes (either .NET or Java) that a mobile application can use to invoke a web service. These classes use the underlying QAnywhere infrastructure to send requests and receive responses. When an object method call is made, a SOAP request is built automatically and

delivered as a message to the server where a connector makes the web service request and returns the result as a message. Authentication is handled on the server side and deserialization is handled on the client side. The application simply has to register a listener and it will be notified when the result has been returned to the client. This provides a reliable and secure method of making web service requests (even while offline and even between application instances) while leveraging the benefits of QAnywhere including sophisticated transmission rules.

#### ADMINISTRATION AND MONITORING

A QAnywhere plug-in to the Sybase Central administration tool provides a wealth of features for both development and ongoing administration.

It features a graphical interface for creating and administering the QAnywhere environment. Support is provided for creating/maintaining client and server message stores, agent configurations, transmission rules, JMS and web service connectors, and destination aliases. The plug-in also allows an administrator to connect to remote message stores, view message history, and browse messages or log files.

#### FACTS AT A GLANCE

##### SERVER PLATFORM:

- Microsoft Windows NT/2000/XP/2003/Vista, Linux, Solaris, AIX, Mac OS X

##### QANYWHERE CLIENT PLATFORMS:

- Windows NT/2000/XP/2003/Vista
- Windows CE/Pocket PC/Mobile