Wireless Application Middleware

Nico de Wet
Dept. of Computer Science
University of Cape Town
ndewet@cs.uct.ac.za

Nadim Yazdani
Dept. of Computer Science
University of Cape Town
nyazdani@cs.uct.ac.za

Bonnie Lam
Dept. of Computer Science
University of Cape Town
blam@cs.uct.ac.za

Ken MacGregor
Dept. of Computer Science
University of Cape Town
ken@cs.uct.ac.za

14 October 2002
Paper number: CS02-11-00

Abstract

The use of middleware has been acknowledged as the principal means of simplifying distributed applications building in the enterprise. Wireless messaging middleware, in particular, allows loosely coupled distributed components and has emerged as being well suited to the wireless environment. In this paper we present a lightweight wireless messaging middleware solution which addresses the reliability and bandwidth issues associated with wireless links.

1 Introduction

Middleware, of which wireless middleware is a specialized subset, has been recognized as an important means of simplifying distributed system construction [Emmerich 2000]. Middleware resolves heterogeneity, and facilitates communication and coordination of distributed components. Since middleware solves a real problem and simplifies distributed system construction, middleware products are being adopted in industry.

Middleware simplifies distributed system construction by addressing the difficulties that arise when building distributed applications. The sources of these difficulties can be broadly categorized into areas of network communication, coordination, reliability, scalability and heterogeneity. Addressing the wealth of difficulties directly using network operating system primitives, and hence without using middleware, is generally too expensive and time consuming.

Wireless middleware, as a specialized subset of middleware, has come into demand due to the proliferation of wireless networks. Additionally, in the last few years, there has been an upsurge in the development of the mobile devices sector. Personal Digital Assistants (PDAs) and a new generation of cellular phones have the ability to access different types of data. With this, the role of wireless middleware has become increasingly important in providing a reliable channel for data access between mobile devices and servers on a wired network.

Wireless middleware is an intermediate software component that is generally located on a wired network, between the wireless device and the application or data residing on a wired network. The purpose of the middleware is to increase performance of applications running across the wireless network by serving as a communication facilitator between components that run on wireless and wired devices [Wireless-Nets 2002]. Wireless middleware serves as a communication facilitator by addressing the numerous ways in which communication can fail between components in a distributed application. Sources of communication failure, amongst many others, include a component going off-line voluntarily but unexpectedly, a component unexpectedly terminating, a component failing to respond to a request in a reasonable amount of time, and communication being severed in the midst of a request [Sundsted 1999].

Typical wireless middleware solutions incorporate intelligent restart and store-and-forward messaging functionality. Intelligent restart is a recovery mechanism that detects when a transmission has been cut, and, when the connection is re-established, resumes transmission from the break point instead of at the beginning of the transmission [Wireless-Nets 2002]. Intelligent restarts are required in order to enhance the robustness of the distributed system that is build using the middleware. A robust distributed system must detect failures, reconfigure the system so that computations may continue, and recover when a link is repaired [Sundsted 1999]. Store-and-forward messaging involves the implementation of message queuing to ensure that users disconnected from the network will receive their messages once the station comes back online.

The basic requirements of wireless middleware can be met by messaging middleware. Messaging middleware serves as a tool for coordinating distributed application components and removes the responsibility for ensuring that messages are delivered reliably and correctly from application components. Messaging products [Softwired 2002; Spiritsoft 2002; Presumo 2002] allow distributed application components to communicate and coordinate their activity (via messages) by providing critical services such as message queuing, message persistence, guaranteed once-and-only once delivery and priority delivery [Silberschatz et al. 1997].

In this paper, a new lightweight wireless middleware solution, henceforth known as Wireless Application Middleware (WAM) is presented. The principal WAM design goals are to simplify distributed application development by focusing on the reliability and bandwidth issues associated with networks that incorporate wireless links. Reliability issues are addressed by

WAM in two ways. Firstly, WAM is based on a subset of the Java Messaging Service (JMS) application-programming interface (API) [Sun 2002a], a message-oriented middleware (MOM) API adopted in industry [Softwired 2002; Spiritsoft 2002]. Secondly, the JMS API is extended to include client message buffering, an intelligent restart mechanism and data compression, the resultant API being known as the WAM-JMS API. The WAM-JMS API is designed to run on both high-end devices connected to broadband-wired networks as well as mobile PDAs running the CE .NET operating system. Bandwidth conservation issues can be addressed by WAM at both the transport and application layer. The development of an optimised wireless transport protocol, which conserves bandwidth, is investigated and text compression is incorporated into WAM.

Section 2 introduces the JMS framework, our development environment (the Microsoft .NET Framework and Compact Framework Beta Version) as well as TCP optimised for wireless links. Related work is outlined in section 3. Addressing reliability issues is discussed in section 4, while in section 5, we describe the project methodology and approach. Section 6 discusses testing and findings. Demonstration applications are described in section 7. Finally, some concluding remarks are made in section 8 and future work is suggested in section 9.

2 Background

2.1 The JMS API

The Java Message Service (JMS) Application Programming Interface (API) allows applications to create, send, receive, and read messages. Messages can arrive asynchronously, meaning the client doesn't have to specifically request messages in order to receive them. In addition, the programmer can specify different levels of reliability depending on the type of message that is transmitted. Unlike RPC, JMS is *loosely coupled* meaning that the sending and receiving applications do not both have to be available at the same time to enable communication [Sun 2002b]. In other words the sender and receiver need not know anything about each other when communicating. This is particularly useful in the wireless domain because of the "sometimes on" characteristics of wireless devices.

A JMS implementation is composed of a JMS Provider, JMS Clients, Messages and Administered Objects.

JMS Provider: This component provides administrative and control features. It is the part of the system that implements JMS interfaces. The J2EE 1.3 platform includes a JMS provider.

JMS Clients: These are components or programs that are the producers and consumers of messages. In the wireless domain clients may include cellular phones and PDAs.

Messages: Messages are the objects that are transmitted between JMS clients.

Administered Objects: These are configured by the administrator, and include destinations and connection factories. Clients use naming and directory services to look up administered objects. For one client to establish a logical connection with another through the JMS provider, it needs to perform a lookup of administered objects in the naming and directory service being used.

JMS supports the two more common approaches to messaging, namely *point-to-point* and *publish-subscribe*. WAM implements the point-to-point model in which each message has only one consumer. Hence only one JMS client may receive messages from a queue at one point in time however any number of clients may post messages to a queue. Messages are posted to a specific queue and are retained until either they are consumed or until they expire. Figure 1 illustrates point-to-point messaging, in which each queue has a single reader but one or more senders.

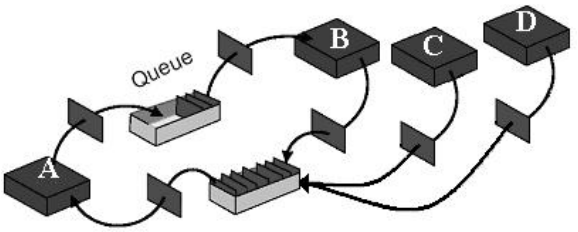

Figure 1: The JMS Point-To-Point Model

JMS includes reliability mechanisms that tackle several issues inherent in message-oriented middleware. The first of these reliability mechanisms is message persistence in the provider, which allows messages to survive provider failures. Message expiration is a second reliability mechanism that may prevent queues filling to capacity in the provider due to inactive clients not reading from their queues. The final reliability mechanism is message acknowledgement. For a message to be considered as having been consumed successfully, it has to be *acknowledged* by the receiving party, which results in the message being deleted in the provider. However, as discussed in section 4, these mechanisms are not sufficient for wireless networks.

2.2 Regular TCP in the Wireless Domain

A major concern in any wireless middleware solution is the limited bandwidth associated with wireless links. JMS clients instantiate a *Connection* object that transparently sets up a TCP connection to the provider. The TCP connection is of concern in wireless middleware since the performance of regular TCP is adversely affected by the presence of a wireless link(s) between the sender and receiver. This problem is being and has been researched extensively.

Wireless links have properties that affect TCP performance. Most importantly they do not provide the degree of reliability that hosts expect. The lack of reliability stems from high uncorrected error rates (or bit-error rates) of wireless links (especially terrestrial and satellite links) when compared to wired links. Additionally certain wireless links are subject to intermittent connectivity problems due to handoffs. Handoffs occur in cellular wireless networks such as GSM and involve calls being transferred between base transceiver stations in adjacent cells.

The properties of wireless links mentioned have adverse effects on the TCP congestion control algorithms [Dawkins et al. 2001]. The root of the problem is that congestion avoidance in the wired Internet is based on the assumption that most packet losses are due to congestion. This assumption is certainly correct in wired links and subnets that have low uncorrected error rates. However, as has been mentioned, wireless links do not enjoy low uncorrected error rates.

The result of the incorrect error-rate assumption is poor TCP performance experienced by users. The reason for this observed poor performance is that TCP incorrectly presumes network

congestion when packets fail to arrive. The sender assumes packet loss (say due to congestion-related buffer exhaustion) and thus substantially reduces traffic levels as it probes the network to determine ”appropriate” traffic levels.

Various recommendations for improving wireless TCP performance exist [Balakrishnan et al. 1996], which can be categorized into link-layer, end-to-end and split-connection solutions as well as header compression techniques [Degermark et al. 1996]. Despite these improvements an important consideration is whether one needs to use TCP in a wireless JMS implementation such as WAM. Industrial wireless JMS implementations [Softwired 2002; Spiritsoft 2002] and other industrial MOM solutions [Broadbeam 2002b; Broadbeam 2002a] use UDP based reliable messaging protocols, which are essentially UDP with a thin reliability layer, as specified in RFC 908.

The argument in favor of using UDP [Bonachea and Hettena 2000] with a thin reliability layer, instead of TCP, is that it typically provides the lowest overhead access to the network and is widely portable. Additionally, when considering wireless links with their limited bandwidth, UDP becomes attractive due to less header overhead associated with each packet. UDP datagrams contain an 8-byte header whereas in TCP implementations one finds 20 to 40 byte headers.

The scope of the WAM system does not allow for the construction of a UDP based reliable messaging protocol and hence regular TCP sockets are used. However the design of the WAM system must be modular to allow for the replacement of the networking module when a UDP based reliable messaging socket becomes available.

2.3 Data Compression

Data compression can be used on messages sent to and received from JMS message queues. This is particularly essential in wireless devices with limited memory and processing capacity, so as to conserve bandwidth. Compressed messages also save storage space in message queues in the JMS provider. Data compression techniques are data type dependant and text compression is incorporated into WAM. The ZIP text compression algorithms [Krueger 2002], which are enhancements of the LZ77 algorithm [Stallings 2000], are used.

2.4 The .NET Compact Framework

The WAM development platform was chosen to be C# using Microsoft Visual Studio .NET. A suitable development language and environment had to be found for an API that runs on Windows CE .NET. Microsoft announced the beta release of the .NET Compact Framework [Microsoft 2002b] in the first quarter of 2002 and subsequently released the beta 1 version of the .NET Compact Framework in July 2002.

The advantages offered by the .NET Compact Framework are significant when developing an application in C# and Visual Studio .NET. The .NET Compact Framework offers a subset of the desktop .NET Framework meaning that developers can reuse existing programming skills and reuse code. Moreover one can migrate portions of existing .NET Framework applications to smart devices, this is because most of the library calls are the same, the language is the same (C#) and the environment is the same (Visual Studio .NET). An additional advantage offered by the .NET Compact Framework installation is an emulator built into Visual Studio .NET which can be used for testing purposes. An API targeting CE .NET would benefit in terms of development efficiency when using the .NET Compact Framework and as such the decision was taken to use this extension to Visual Studio .NET in the WAM project.

2.5 Distributed Application Building Technologies

Contemporary distributed applications are rarely built from the ground up using network operating system primitives (sockets). Instead technologies such as JNDI (Java Naming and Directory Interface) [Sun 2002c], RMI (Remote Method Invocation) [Sun 2002d], ADSI (Microsoft Active Directory Service Interfaces) [Microsoft 1999] and Remoting [Liberty 2001] are used. Similarly data sent through the network is not normally sent in an unstructured format, as happens when object serialization is used before transmitting objects through a network. The technologies mentioned are used in open-source JMS implementations and give insight into the possible ways of constructing the WAM system.

3 Related Work

JMS provides a viable alternative for communicating in the wireless domain where unreliable connectivity is expected. Traditional JMS was designed to allow Java applications to communicate with existing wire-line MOM systems. As a result, a full JMS implementation is too ”fat” for wireless devices because low power consumption and a smaller memory footprint are required [SpiritSoft 2001]. Two primary wireless JMS products exist and they make use of lightweight JMS libraries in order to function effectively.

3.1 SpiritSoft - The SpiritArchitecture

SpiritSoft has developed a commercially available JMS implementation, namely SpiritLite, which is specifically designed for use by mobile devices. SpiritLite offers a small footprint by making use of SpiritSoft’s Java LightWeight Message Service (JLWMS), a stripped-down version of JMS. In addition JLWMS makes use of a lightweight wrapper around a JLWMS message payload thereby significantly reducing the overhead associated with JMS messages (300 bytes per message even when empty). SpiritLite also allows serverless messaging between clients and gives the client the option to use UDP transport instead of TCP when reliability is not an issue.

3.2 Softwired - iBus//Mobile

The iBus//Mobile product from Softwired makes use of Wireless JMS (WJMS), a lightweight implementation of JMS. The design goals of iBus//Mobile are similar to that of SpiritLite in that a small footprint is offered. WJMS is capable of implementing both point-to-point and publish/subscribe in a Java library of only 70k, and at run-time an iBus//Mobile applications requires as little as 50k heap space [Maffeis 2002].
IBus//Mobile is able to implement WJMS and still adhere to the JMS message formats (as specified in the official JMS specification [Sun 2002a]) by using a Mobile JMS Gateway that sits in between the clients and the JMS Provider (a J2EE Provider), as shown in Figure 2. To the JMS provider, the gateway appears to be a regular JMS client. From the client’s point of view, however, the gateway acts as a communications hub and message format translator. The gateway guarantees delivery of messages to the receiving party.

The WJMS client library is intended for deployment on programmable wireless devices, but the iBus//Mobile architecture also caters for non-programmable devices such as pagers and cellular

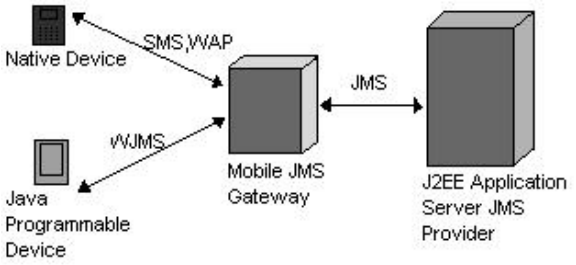

Figure 2: The iBus//Mobile Architecture

phones, as shown in Figure 2. Therefore it could be said that the client library is optional and depends on the nature of the client device.

4 Addressing Reliability Issues

As described in Section 2, JMS supports both the point-to-point and publish/subscribe messaging paradigms. For the purposes of this investigation we will focus primarily on point-to-point messaging. Since publish/subscribe essentially uses the underlying queue infrastructure used in point-to-point messaging, we feel it important to focus on developing a robust infrastructure that would initially support point-to-point, and be easily extensible to include publish/subscribe when required.

We have decided to implement a subset of the JMS architecture. Because our primary intention is to investigate reliability in the wireless domain using C# as the development environment, implementing the entire JMS specification in C# is both unnecessary and beyond the scope of this investigation. Having mentioned this, there exist a few areas where a 'pure' implementation of JMS (which has been developed mainly for wired networks) would prove unsatisfactory in the wireless environment. Our implementation aims to address these issues.

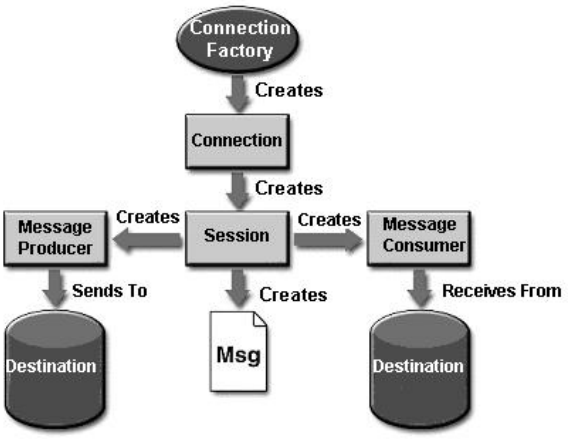

Figure 3: The JMS Programming Model

We intend to maintain the original JMS class hierarchy, shown in Figure 3. The Connection Factory and Destination (Queue) classes would both perform lookups using a basic naming server. Sessions will use the Auto-Acknowledge mechanism, meaning that all messages received from the provider need to be acknowledged. Sessions are also non-transactional by default. The Message Consumer receives messages synchronously, and the user of the API can decide between blocking and non-blocking synchronous receives. The Message Producer sends messages to a queue at the provider and then waits for an acknowledgement. If the acknowledgement fails to arrive in time then our **reliability mechanisms**, discussed later in this section, will take effect.

The Message class will initially support text messages only but JMS is modular enough to allow other message types to be plugged in easily. We have included the JMS message header as is, however because not all message header fields are necessary for our purposes we will provide support for the following:

- JMSDestination
- JMSDeliveryMode
- JMSReplyTo
- JMSType
- JMSRedelivered

We intend to provide added functionality to JMS in order to satisfy the requirements of wireless messaging. More specifically, our implementation will address the issues mentioned in the Introduction:

4.1 Intelligent Restarts

If the provider fails to acknowledge a sent message or a connection is cut at any time, a reconnection method is called to try to re-establish the connection transparently to the client. In addition, the WAM provider will maintain state on behalf of the disconnected client in order to reduce the overhead of re-establishing the connection and session details. A client connected to the Provider on a low bandwidth wireless link will thus benefit by saving time that would have to be taken to restore its state in the Provider. Naturally, if reconnection fails after a specified time period the provider will cease to maintain the client's state.

4.2 Low Bandwidth

We will investigate the use of message compression to decrease the amount of data that needs to be transmitted over low bandwidth wireless links. To put compression into practice we will extend the Message class to include compression and decompression methods, and the message header to include a JMSCompressed flag. A client wishing to compress a message would call the compression method and set the JMSCompressed flag. At the receiving end the JMSCompressed field would be read and the message decompressed if necessary.

Furthermore, bandwidth constraints will be reduced by not serializing the Message class before transmission across the wireless network. Object serialization produces metadata which increases overall packet size. Having mentioned this, we have discovered that object serialization is not supported in the Windows CE .NET Compact Framework Beta Version, as confirmed by Ginger Staffanson of Microsoft [Staffanson 2002].

4.3 Store-and-Forward Messaging

This is intrinsic to the WAM framework. The provider stores messages in queues and also logs them to disk if they are specified as being persistent. In addition, the client implements a message-queuing mechanism whenever the connection to the provider fails and messages are still being sent by the user. Once a connection is re-established the queued messages are delivered to the provider.

5 Methodology and Approach

The WAM Project followed an incremental *analyse-implement-redesign* methodology; in other words each part of the system was analysed, coded, tested and if necessary redesigned. Analysis of the system was performed using the Unified Modelling Language (UML). In the initial phases, Use Case diagrams and their corresponding narratives were produced, and these were subsequently refined as the project evolved. Sequence diagrams were also produced to model the inter-class communication required for achieving specific tasks. Using these diagrams and with the aid of the JMS Specification, we arrived at a conceptual class diagram for the WAM-JMS API. In addition we abstracted out the components that would be required for a functional system. The components and the functions which they are required to perform are listed as follows:

1. **A Naming Server**, which maintains information on the location of WAM Providers and the existence of queues. It must support three basic sets of operations on this information, namely adding, deleting and performing lookups. It was felt that the relatively small size of our infrastructure did not warrant the use of a large scale Naming and Directory Service (NDS) such as ADSI. Moreover only a basic set of services is required and we could reduce unnecessary overhead by catering for these services only. Thus a lightweight Naming and Directory Service, known as WAM-NDS, would be developed with the aim of providing efficiency and the lowest overhead for our basic set of operations.

   WAM-NDS would use UDP as its transport layer protocol. Since WAM-NDS operates as a simple request-response service there is no need to use a stream-based, connection-oriented protocol such as TCP. The connectionless, unreliable UDP is sufficient for our purposes and makes WAM-NDS operations occur considerably faster than if TCP were used.

2. **An Administrative Console** for adding and deleting the information in the WAM-NDS. The administrative console may conceptually be located anywhere on the network so it must be designed with remote access capability. It has been decided that for the purposes of this project, the WAM provider will contain an embedded administrative component. This is simply done for convenience in order to reduce the number of standalone components in the infrastructure.

3. **The WAM Provider** is the central core of the entire infrastructure. The message queues physically reside at the Provider. When the Provider starts it must register its presence with the Naming Service. The Provider will require some mechanism to keep track of details pertaining to each queue. These details include knowing which client is reading from a queue, restricting readership of a queue to one client only and knowing how many messages are in the queue at any time. The Provider should be able to dynamically create temporary queues and delete them when the client disconnects. It must also provide message persistence if the message header specifies this.

   The Provider will also need to maintain information about each client. A unique ID is assigned to each client upon connection establishment and serves as a means by which the Provider will identify individual clients. The Provider should maintain state on behalf of the client in case a connection is dropped unexpectedly. It should therefore be able to 'rebuild' the connection parameters if a client reconnects within a specified time period.

4. **Two WAM-JMS APIs**, one for a regular wired network where regular disconnections and low bandwidth are uncommon, and another specifically tailored for the wireless link. Compression capability will be supported in both versions but intelligent restarts and client-side message queuing is specific to the wireless API.

Implementation of the various components was carried out using Microsoft Visual Studio .NET and C# in particular. The WAM-NDS was the first to be constructed since the other components are reliant on a Naming Service during initialization procedures. Both the point-to-point and publish/subscribe messaging paradigms are supported by the WAM-NDS, hence it can be used in similar investigations on publish/subscribe messaging. WAM-NDS maintains two information bases. One concerns WAM Providers and their network location while the other involves confirming the existence of Destinations.

Next the generic (non-wireless) API and the WAM Provider were developed in parallel. The development of the components proceeded iteratively. The modularity of the API and the ordered sequence in which classes must be executed in the API eased the development process to some degree. The JMS Specification places restrictions on the order in which classes are created - for example, a ConnectionFactory object must be created in order to create a QueueConnection, which must be created in order to create a QueueSession, and so on (Figure 3).

The Provider is composed of two interacting components - a networking subsystem and a message management subsystem. The networking subsystem contains a multi-threaded TCP/IP server which receives packets from clients and passes them to the message management subsystem. The packets are read, interpreted and acted upon by the message management sub-system.

An Administrative Console has been incorporated into the Provider. The console allows the WAM Administrator to add and delete queues from the WAM-NDS and Provider. It also supports commands specific to the Provider, such as viewing the details of connected clients, dropping client connections and viewing queue details. The console uses a command-line interface similar to the JMS Provider in J2EE (Java 2 Enterprise Edition) and most of the commands are syntactically similar. The Administrator also has the option of using a menu if this is preferable.

The next step was to devise a suitable mode of communication between the WAM-JMS API and WAM Provider. Ideally a distributed object technology such as RMI would fulfil our requirements. The .NET framework's distributed object mechanism, known as Remoting, simplifies distributed application building. Knowing that object serialization is not supported by the CE. .NET Compact Framework, it would seem unlikely that Remoting be available. Our assumptions were confirmed by Windows CE Program Manager Alex Yakhin: *"The remoting functionality is not included in . NetCF. Your best bets would be Sockets or WebServices"* [Yakhnin 2002].

We therefore shifted our attention to a lower-level alternative - designing an application protocol using Sockets. This effectively meant building a protocol "from the ground up". Although application protocols are said to be cumbersome and potentially error-prone [Sun 2002d], they nevertheless prove effective if implemented carefully as we have discovered. Furthermore, because the messages making up the protocol are short, delimited strings they provide very little overhead - an advantage on slow wireless links. All communication between the client APIs, the WAM Provider and the WAM-NDS is achieved by means of

application protocols. The strings which make up the API-Provider protocol are of the general form:

[client ID]":"[code]":"[rest of message]

The client ID identifies the client to the Provider; the code specifies the exact type of the message and the rest of the message consists of one or more comma-delimited fields.

Example: *2324:7:billing,,n,*

In this case, 2324 is the Provider-assigned client ID, 7 is the message code corresponding to the API requesting a message from a particular queue, billing is the name of the queue and n means that the call is synchronous and non-blocking (the API implements a timeout mechanism in case no messages are in the queue). The Provider replies with the following if a message is in the billing queue:

Example: *2324:0:[WAM-JMS message],*

Code 0 corresponds to a normal WAM-JMS message. The WAM-JMS message consists of comma-delimited header fields (see Section 4) followed by the message body.

The next phase of development involved incorporating the reliability mechanisms, described in Section 4, into the WAM System. The sending and receiving methods of the generic API were adjusted to detect disconnections from the Provider and attempt to reconnect. The Provider was enhanced with a client state-saving mechanism to allow reconnected clients to carry on where they had left off before the disconnection. More specifically, the Provider restores any temporary queues created by the client, and re-registers the client as a reader for queues which the client was reading from previously.

Persistence capability was also added to the Provider. Messages that arrive at the Provider with the JMSDeliveryMode header field set to persistent are immediately serialized to permanent storage in case of an unforeseen Provider failure. In the event of failure, when the Provider reboots it will recreate all administratively created queues and repopulate these queues with transient message headers. The transient message headers serve the purpose of pointing to the persistent messages in the Provider's persistent store. In addition, by putting only the message headers of persistent messages in queues, the Provider is able to conserve memory. The usefulness of a persistence mechanism is illustrated in the demonstration application of Section 7.

Support for temporary queues was incorporated into the Provider and client APIs. This allows queues to be created "on the fly" by client applications. Temporary queues prove particularly useful when an API does not have access to a permanent queue, but would nonetheless require a response from some backend application after enqueuing a request.

Finally, a compression mechanism was included for text messages that exceed an experimentally-determined threshold size. The compression library used was *SharpZipLib* an open source C# translation of the full Java *zlib* library, originally developed by the *Free Software Foundation (FSF)* for text message compression [Krueger 2002]. Although only text compression was implemented, the modularity of JMS allows other message types to be easily extended with their own compression schemes.

6 Testing and Results

WAM system testing was done iteratively. Separate tests were firstly conducted on the Provider, the general (desktop) WAM-JMS API, the CE.NET WAM-JMS API and text compression before integration tests were conducted. It should be noted that correctness rather than performance testing on the WAM system as a whole was more feasible. Due to the unavailability of a wireless network and limited availability of specific software needed for the project, we could not conduct load testing. For example, it was not possible to test whether a large number of mobile clients connected to the Provider would degrade Provider performance. Also, testing of transmission speed was not meaningful, as TCP was used as the transport protocol in a wired ethernet network, and indeed, transmission of packets would be much faster than a wireless network.

6.1 Provider Testing

Unit testing on the Provider initially consisted of writing a program to test the Provider's responses to client's requests, and checking whether its responses conformed to the message formats in the WAM application protocol. Initial tests included testing single requests, a sequence of requests and the menu presented on the administration console. Coding errors found were corrected before continuing to the next Provider testing phase.

The Provider was again tested after the additions of text message compression and message persistence. For example, the tests performed initially, namely, testing single requests, multiple requests and the administration console were repeated. These tests were repeated to ensure that additions to the scope of the services provided by the Provider did not introduce adverse effects. Results from these tests showed the Provider responded as expected.

6.2 General API Testing

Testing the general (desktop) WAM-JMS API involved writing different test applications to test each aspect of the API. As new classes were added to the API, the test application(s) were modified to test these added classes, with specific emphasis on exception cases.

The primary test application was a console chat program. The chat program was developed hand-in-hand with the API. As functionality was being added to the API, the chat program was extended to include this functionality and at the same time test that it worked as expected. Later on, when support for temporary queues was included, another application was developed for testing purposes.

6.3 CE.NET API Testing

Two areas in the CE.NET API required testing, namely, splitting string packets and reconnection with the Provider. Because of the limitations of the .NET Compact Framework, a PacketSplitter class had to be written to split the JMS message correctly. Reconnection with the Provider was implemented in the CE.NET API due to the high likelihood of frequent connection cuts when used in a wireless network.

A few scenarios were used to test reconnection with the Provider, for example, dropping the connection when the client program was busy sending or receiving messages. To simulate a disconnection between the mobile device and the Provider, the administration console embedded in the Provider is used to close the client's TCP connection. Results from the tests indicated that reconnection

specified in WAM application protocol was correctly followed and that the client application reconnected with the Provider successfully. In addition, the Provider correctly maintained the client's state when a successful reconnection occurred.

6.4 Initial Compression Testing

Initial compression testing was conducted to determine the length at which text compression was worthwhile. Strings used for testing ranged from 0 characters to 960 characters, some including repeated patterns. All available *SharpZipLib* compression levels were used to compress these strings. Compression levels 1- 3 were for fast compression and 4-9 for optimum compression. After analysing the tests, it was decided that string messages of 300 characters or longer were compressed, because they yielded a compression ratio of 1.5 and above. It was also decided the default compression level was set to 3 for fast compression. Fast compression was chosen over optimum compression because optimum compression levels only yielded a compressed string that was about two bytes fewer than when using fast compression. For the WAMsms application, it was decided that SMS messages longer than 160 characters would be compressed to simulate real-life extended SMS messages.

6.5 Findings

To conclude, the WAM system functioned correctly as specified. Despite the limitations inherent in the .NET Compact Framework, such as lack of support for object serialization or Remoting, the WAM system has proven that successfully developing a wireless distributed system is still possible.

7 Demonstration Applications

To demonstrate the usefulness of WAM-JMS in distributed application building, two prototype applications have been developed. The first, a Graphical User Interface (GUI) chat application called *WinChatta*, was developed during the testing phase and further refined as modifications were made to the WAM framework. The application is designed to enable chats between individual clients; however it could easily be extended to allow several parties to communicate simultaneously. Unlike most other peer-to-peer chat applications, *WinChatta* supports the sending of messages while one party is offline. This "loose coupling" of communicating clients is where the power of messaging middleware lies.

The second application is intended to demonstrate a more complex scenario where WAM-JMS could be used commercially, and is discussed in more detail below.

7.1 WAMsms - Extended SMS Service

WAMsms is a general-purpose Short Message Service (SMS) application incorporating persistent messages and billing information. The prototype application is built entirely over the WAM-JMS framework and illustrates a scenario where WAM-JMS can be used successfully in a commercial setting. The components of the system are as follows:

- Clients - running the *WAMsms* application.
- The WAM-JMS Provider - holds message queues corresponding to the client's phone number as well as billing information.
- A Billing sub-system - retrieves billing information from the Provider on a regular basis and calculates total costs and message statistics for each user.

The user may send extra-long SMS messages, up to a maximum of 8192 bytes (the maximum WAM-JMS message size). Messages exceeding a certain size threshold are compressed before transmission, as described in Sections 4 and 5.

A typical interaction scenario involves several communicating clients, all transparently sending persistent billing information to the Provider. The billing information is immediately written to a persistent store at the Provider in case of a system crash. As a result, the company running the SMS service is *guaranteed* not to lose money in spite of a system failure. At the end of each day, the billing sub-system performs a batch download of the billing information and presumably incorporates this information into a backend database (not implemented in this prototype). Figure 4 provides a conceptual representation of this scenario.

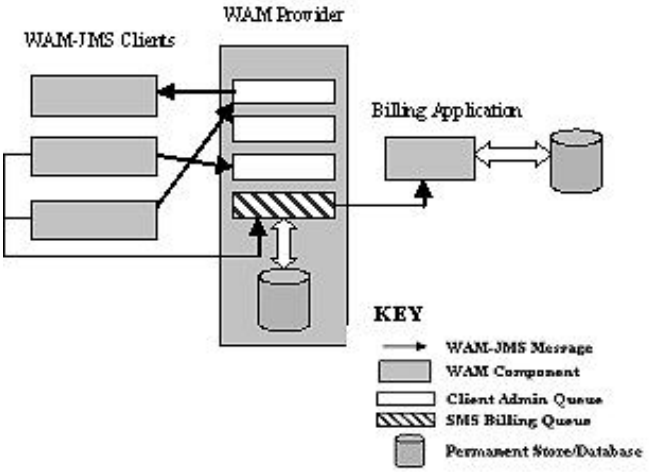

Figure 4: WAMsms Conceptual Diagram

Every time an SMS is sent from one client to another, billing information is sent to the SMS Billing queue. Billing messages are made are automatically specified as being persistent. The Provider, on discovering that a message should be persistent, serializes the message to a persistent store and adds it to the receiver's message queue. Normal SMS messages are by default transient however the application developer can choose to enable persistence if required.

The user of the WAMsms application is initially prompted to enter a login ID (or mobile phone number as the case may be). The application goes on to perform a lookup of the Provider, client admin queue and billing queue at the WAM-NDS server. It then establishes a connection with the Provider. Messages that were stored in the queue arrive one by one and the sender's mobile number is subsequently displayed in the Inbox. The user can now read the newly-arrived SMS messages, including extended messages.

Figure 5 shows the user about to send an SMS message. After the user has entered the receiver's mobile number, two messages are sent to the Provider; the first being the message itself and the other being the persistent billing message (sent to the SMS Billing queue). The billing message contains the name of the sender in the JMSReplyTo header field, and the body merely contains the number of characters contained in the sent SMS. The latter piece of information determines the billing structure in our prototype however the application designer is free to include any data which may affect billing, such as the time sent and the user's contract type.

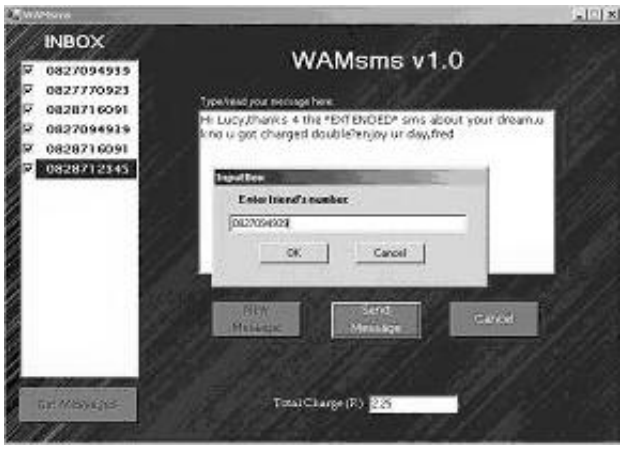

Figure 5: The WAMsms User Interface

The Billing Service in the WAMsms prototype is designed for batch processing rather than continuous operation, although once again the application designer is free to decide which model is more suitable. The billing sub-system retrieves messages from the SMS Billing queue at an 'off-peak' time in order to maintain the efficiency of the Provider. The Billing Service uses this billing information to separate normal and extended SMS messages, and applies different charges to each type of message.

8 Conclusions

Wireless middleware is a relatively new field in distributed computing. However, with the great advancements made in the field of mobile computing, the need to develop wireless middleware to facilitate communication between wireless applications becomes apparent.

The goals of the Wireless Application Middleware (WAM) project were firstly, to perform research on the challenges associated with developing wireless middleware and secondly, to devise a solution that would resolve these challenges to some degree. Our implementation, the WAM System, made use of a subset of the Java Message Service (JMS) API [Sun 2002a], in particular, the point-to-point model. The JMS API is a widely accepted industry standard for developing Message-Oriented Middleware (MOM). However, the JMS API is not specifically intended for developing wireless MOM. Thus, to fulfil the basic requirements of a wireless middleware API, the JMS API has been extended in the WAM-JMS API to include message persistence, intelligent restarts and data compression.

One of the challenges identified in developing wireless middleware and JMS in particular, was that regular TCP is not an ideal transport protocol when used in wireless networks. Initially, we proposed to develop an optimised version of TCP for wireless middleware. The research phase of the project showed that a reliably-delivered messaging (Rdm) socket based on UDP, as opposed to an optimised version of TCP, would suit the requirements of the WAM system to a greater degree than optimised TCP. However, the development of a Rdm socket based on UDP was beyond the scope of the project and thus regular TCP was chosen as the transport protocol despite known limitations when used in wireless networks. Consequently, the focus of the project was shifted to further research on designing and implementing the API and the service Provider.

We have implemented our own Naming and Directory Service *(WAM-NDS)*, the WAM Provider and two client APIs, one for the desktop and the other for the CE .NET operating system (found in wireless devices). Transparent reconnection with the WAM Provider is implemented in the CE.NET version of the WAM-JMS API and message persistence is managed by the WAM Provider.Two end-user applications,*WinChatta* and *WAMsms* were developed for both environments to demonstrate the usefulness of the WAM API. Data compression was incorporated into both applications.

Due to the unavailability of a wireless network, correctness testing, rather than performance testing on the WAM System was more feasible. Both unit and integrated testing showed the correctness of our WAM implementation. In addition, the effectiveness of intelligent restart mechanisms was demonstrated by the transparent reconnection functionality incorporated into the CE .NET WAM-JMS API.

In conclusion, we have addressed some of the basic requirements of wireless middleware and have demonstrated the usefulness of the WAM System. The example applications developed using the WAM System have made it clear that messaging middleware, such as WAM, greatly reduces the complexity of seemingly difficult operations by providing a decoupling layer between clients and servers. We hope that the WAM Project will serve as a starting point into further research in this field.

9 Future Work

Various areas in WAM could be considered for extensions. The most pertinent extensions are considered here.

9.1 Benchmarking the WAM system

Benchmarking the WAM system when a wireless network is available would provide useful information for a deployment strategy. Ideally one would conduct these benchmark tests using industrially established metrics. A useful starting point for an investigation into benchmarking is documents supplied by Sonic Software [SonicMQ 2002].

9.2 Provider and the NDS Application Types

The WAM Provider and the WAM-NDS server are currently both console applications. It would be desirable to convert these to Microsoft Windows Service Applications. Microsoft Windows Services are intended to be used for long-running executable applications that run in their own Windows sessions. In other words, the services run without the need for a particular user to be logged on [Microsoft 2002a]. Services have no user interface and can be automatically started when the computer boots. Services can be started, stopped and paused at will.

9.3 Overall API Extensions

Security is one important aspect for which the WAM-JMS API could be extended. For example, mechanisms for authentication could be included when a client wishes to register as a reader of a queue. The client could send a user name/password pair to the WAM Provider for authentication, before registration proceeds.

The WAM-JMS API could also be extended to support other types of JMS message types such as the JMS *ByteMessage*. A

new *ByteMessage* class could be implemented to support file transmission. The *JMSType* field would be used to indicate the message type. Extending the API to support multiple message types implies extending the compression library accordingly. While sending JMS messages, the client would check for the *JMSType* and use the appropriate compression algorithms and compression levels for that message type.

9.4 CE.NET API Extension

Apart from the CE.NET API initiating reconnection with the WAM Provider, another useful mechanism to include would be message persistence in the API. It is known that the battery life of mobile devices is very limited, so making messages persistent would guard against battery failure. In order to provide some form of message persistence, one would have to detect the battery level of the PDA. The Power properties in Windows CE.NET allow the API programmer to obtain information about the battery level. The programmer can then choose to make messages persistent once the battery level falls below a certain percentage.

References

BALAKRISHNAN, H., PADMANABHAN, V. N., SESHAN, S., AND KATZ, H. 1996. A comparison of mechanisms for improving tcp performance over wireless links. In *ACM SIGCOMM '96*.

BONACHEA, D., AND HETTENA, D., 2000. Amudp: Active messages over udp. {bonachea,danielh}@cs.berkeley.edu.

BROADBEAM, 2002. Axio, broadbeam's mobile software platform. Internet. Available: http://www.broadbeam.com/pdf/axio_white_paper.pdf.

BROADBEAM, 2002. Expressq. Internet. Available: http://www.broadbeam.com/pdf/expressq.pdf.

DAWKINS, S., MONTENEGRO, G., KOJO, M., MAGRET, V., AND VAIDYA, N. 2001. End-to-end performance implications of links with errors rfc 3155.

DEGERMARK, M., ENGAN, M., NORDGENAND, B., AND PINK, S. 1996. Low-loss tcp/ip header compression for wireless networks. In *ACM MobiCom*.

EMMERICH, W. 2000. Software engineering and middleware: A roadmap. In *Proceedings of the conference on The future of Software engineering*, ACM Press, New York, USA, 117–129.

KRUEGER, M., 2002. #ziplib the zip, gzip, bzip2 and tar implementation for .net. Internet. Available: http://www.icsharpcode.net/OpenSource/SharpZipLib/.

LIBERTY, J. 2001. *Programming C#.* O' Reilly, July.

MAFFEIS, S., 2002. Jms for mobile applications and wireless communications. Internet. Available: www.softwired-inc.com/people/maffeis/articles/softwired/profjms_ch11.pdf.

MICROSOFT, 1999. Adsi open interfaces for managing and using directory services - white paper. Internet. Available: http://www.microsoft.com/windows2000/docs/adinterface.doc.

MICROSOFT, 2002. Introduction to windows service applications. Internet. Available: ms-help://MS.VSCC/MS.MSDNVS/vbcon/html/vbconintroduction-tontserviceapplications.htm.

MICROSOFT, 2002. .net compact framework overview. Internet. Available: http://msdn.microsoft.com/vstudio/device/compactfx.asp.

PRESUMO, 2002. Internet. Available: http://www.presumo.com.

SILBERSCHATZ, A., KORTH, H., AND SUDARSHAN, S. 1997. *Database System Concepts*. McGraw-Hill.

SOFTWIRED, 2002. Internet. Available: http://www.softwired-inc.com.

SONICMQ, 2002. Benchmarking e-business messaging providers (article). Available: http://www.sonicsoftware.com/products/how_to_benchmark.htm.

SPIRITSOFT. 2001. Jms: Extending the enterprise to real time wireless messaging. *SpiritSoft Technical White Paper*.

SPIRITSOFT, 2002. Internet. Available: http://www.spiritsoft.com.

STAFFANSON, G., 2002. Net 247 newsgroup - remoting and .net cf? Internet. Available: http://www.dotnet247.com/247reference/msgs/12/61219.aspx.

STALLINGS, W. 2000. *Network Security Essentials - Applications And Standards*. Prentice Hall, New Jersey, USA.

SUN, 2002. Java message service specification version 1.1, April.

SUN, 2002. Java message service tutorial. Internet. Available: http://www.http://java.sun.com/products/jms/tutorial/1_3_1-fcs/doc/overview.html, April.

SUN, 2002. Java naming and directory interface. Internet. Available: http://java.sun.com/products/jndi/.

SUN, 2002. Java remote method invocation specification. revision 1.8, java 2 sdk, standard edition, v1.4. Internet. Available: http://java.sun.com/products/jndi/, September.

SUNDSTED, T., 1999. Messaging makes its move. Internet. Available: http://www.javaworld.com/javaworld/jw-02-1999/jw-02-howto.html, February.

WIRELESS-NETS, 2002. Wireless network middleware. Internet. Available: http://www.wirelessnets.com/articles/whitepaper_middleware.htm.

YAKHNIN, A., 2002. Google groups (group:microsoft.public.dotnet.framework.compactframework). Internet. Available: http://groups.google.com/groups?hl=en&lr=&ie=UTF-8&group=microsoft.public.dotnet.framework.compactframework.