A DATA DISTRIBUTION SYSTEM FOR MOBILE DEVICES

By

Jonathon Williams

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Computer Science

December, 2011

Nashville, Tennessee

Approved:

Dr. Theodore Bapty

Dr. Sandeep Neema

To my parents, for all their support and prayers through the years.

ACKNOWLEDGMENTS

This work would not have been possible without the support and guidance of my advisor, Dr. Theodore Bapty. I would also like to thank Dr. Sandeep Neema, for his guidance and advice through the design and implementation of this project.

I would like to give a special thanks to my parents and to my sister, Brenna, for all their prayers, love, and support through my academic career. I would also like to give a special thanks to my undergraduate professor, Dr. Greg Nordstrom, without whom I would not have found my way to Vanderbilt University and the Institute for Software Integrated Systems and would not be working on this project now.

Financially, this project was made possible through a grant from DARPA's Transformative Apps program.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

One major trend recently in consumer electronics has been the smartphone– a compact, powerful, networked computing device with continuous networking connectivity and the ability to run a variety of mobile applications. These applications, often referred to as "apps" in popular media, provide the user with access to dynamic information while away from home or the office.

With this new emphasis on dynamic mobile applications, an efficient way to distribute data between devices and the services which provide information to them is needed. Since smartphones have limited battery capacity and typically run on relatively low-bandwidth cell networks, such a data distribution system should be focused on minimizing power and bandwidth consumption. Also, smartphone users expect up-to-date information to be available immediately when they open an application, so such a framework should allow applications to receive needed data at any time, even when the user isn't actively using the application. This paper describes such a data distribution system, called AMMO (Android Mobile Middleware Objects), focusing specifically on its server-side component, the AMMO Gateway.

I.1 Motivation

AMMO is being implemented as part of DARPA's Transformative Apps program, which is an effort to bring the power and mobility of modern smartphone platforms into the hands of Army soldiers on the battlefield. AMMO forms the core middleware on which all the other mobile applications in the Transformative Apps program will be based. Therefore, the AMMO project is currently focused on assisting in the development of military applications. These applications include simple, commonplace applications such as instant

messaging or photo sharing, but the main driving force behind AMMO and the gateway is situational awareness. This entails providing the soldier with a picture of the battlefield: positions of fellow soldiers and of enemy combatants, hazards and terrain features in the area, and other information the solder might need to know in order to do his job effectively.

Mobile devices, such as the Android phones which are targeted by the Transformative Apps program and AMMO, are particularly well-suited for this application. Their mobility gives each soldier almost-continuous access to the information they need to know, right when they need to know it. Modern phones are also equipped with an array of sensors, such as GPS to provide real-time location updates to the solder, or cameras to allow a soldier to provide a visual record to go along with reports from the field. These phones can also be equipped to provide ubiquitous connectivity, through technologies such as WiFi or 3G cell service, or through more secure military radios. This could allow real-time information sharing between soldiers in the field.

AMMO and the AMMO gateway provide the glue to hold these military apps together. Although we would like for AMMO to be as broadly useful as possible outside of military applications, this specific application is the driving force behind all of AMMO's goals:

Power efficiency: While in the field, a soldier may be away from a charging station for days at a time. Mobile applications should consume as little power as possible to allow the device to remain useful for as long as possible.

Bandwidth efficiency: In most areas where AMMO might be deployed, fixed network infrastructure is very limited. This means that users will typically be connected through low-bandwidth radios or satellite connections. Mobile applications should be aware of the limitations of the network connection they are using, and should use bandwidth sparingly to ensure prompt delivery of data and to maximize quality of service for other users.

Reliable delivery: AMMO is expected to be able to handle many types of data, and some

of that data may be mission-critical. For example, one type of data which AMMO could handle are medical evacuation requests: if the request is not delivered promptly, the injured person's condition could deteriorate. Requests also need to be delivered reliably and provide acknowledgements of delivery, so that the requestor can take appropriate action in the event that the lack of network connectivity prevents message delivery. Applications need to ensure that data is delivered as promptly and completely as network conditions allow, and should also ensure that mission critical data is given priority over less important traffic.

Security: In the field, a networked application could be subject to any number of different hostile attacks. Applications should ensure that data is secure and can only be viewed and modified by authorized personnel, and should ensure that the network itself is robust and can remain operational while being disrupted due to outside interference.

CHAPTER II

DESIGN

II.1 Gateway

In the current AMMO design, applications and services do not communicate directly with each other. Instead, they communicate with a common back-end server called the Gateway. The middleware running on each device creates a single connection to the gateway, which is used to asynchronously send and receive push messages (as described in section II.2). In this model, each application does not make its own connections to servers or other devices. This allows for a single point of control to optimize bandwidth utilization, quality of service, and power consumption: these resources can be used more efficiently if all applications are cooperating to exchange data.

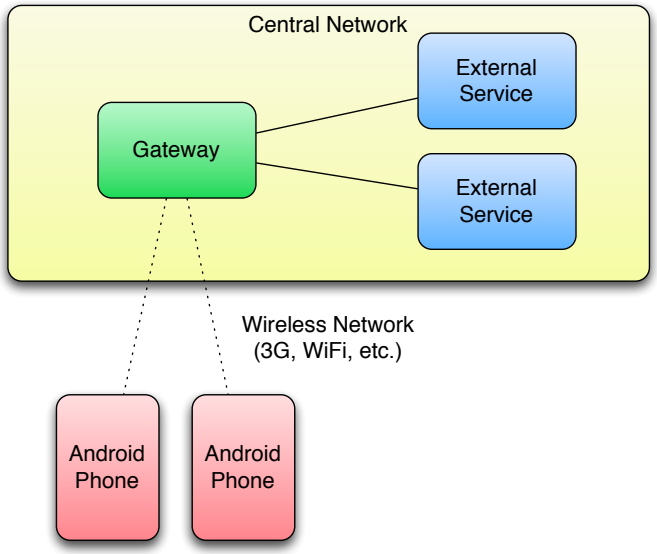

Figure II.1: AMMO network layout

In our current implementation, the gateway sits on a standard server platform (currently, an Intel-based server machine running Red Hat Enterprise Linux) in a data center operated

by the service provider providing services based on AMMO. Mobile devices typically connect to the network via some type of wireless network; depending on the application, this could be a WiFi network, a 3G cellular network, or some kind of packet radio system. This architecture is shown in figure II.1.

By themselves, the gateway and devices are of limited utility; the gateway provides a good way for devices to communicate amongst themselves, but it doesn't provide a way for devices to interact with the rest of the world, using software and protocols outside our control. Therefore, the gateway needs a way to interact with third-party code. This third-party code is connected to the gateway via a plugin mechanism, which allows the gateway to host lightweight components which can interact with other components and services. For example, a developer might want to use the AMMO system to provide a messaging service to users, allowing devices to communicate with each other and with users on other types of devices and other platforms. This developer could write a gateway plugin to allow devices connected to the gateway to interact with an existing messaging protocol such as Jabber, or with a web service which archives messages that are sent to and from devices.

For stability and security reasons, these gateway plugins operate as separate processes from the gateway core itself. They communicate with the core gateway via local inter-process communication (much like a device would communicate with the gateway, except that plugins typically reside on the same physical machine as the gateway core). A plugin developer doesn't have to be concerned with the out-of-process nature of a gateway plugin, however; a clean plugin API is provided (called LibGatewayConnector) which abstracts all of the low-level communication and management details from the plugin developer. This API is described in section II.4.

II.2 Gateway Operations

The AMMO gateway currently supports two different types of communication between devices and the gateway: publish-subscribe communication, and pull-response communication.

Publish and subscribe communications implements the same type of publish/subscribe model as implemented in systems such as DDS (the Data Distribution Service). In this model, a number of clients "subscribes" to receive a particular type of data that they're interested in, referred to by MIME type. Then, when another client "publishes" data of that MIME type, the gateway will automatically send it to all the clients which have subscribed to that type. This is useful for any situation where a number of devices might be interested in receiving all data of a particular kind or on a particular topic, as one might encounter in an application like real-time chat.

In pull/response messaging, a client makes a query (or a "pull request") for a particular type of data which matches parameters specified by the client. Services can register to handle pull requests of specific types, and when a pull request is received, the request is forwarded on to all the services which have registered to handle pull requests of that type. The services then send matching data back to the gateway, which transmits the results back to the originating client. Pull/response messaging is useful when a device needs to receive data on-demand, but doesn't need continuous updates like in publish/subscribe messaging.

Both types of messaging are designed to operate asynchronously. For example, a client can continue to do other things, such as interacting with the user or publishing more data, while it is waiting for data to be published, or waiting for data to be returned from a pull request. This is important because a device may have transient connectivity, or may be in a limited bandwidth or high-latency environment. In these environments, operations may take a long time to complete, so it is critical that devices remain useable while AMMO is completing an operation.

II.3 Networking Layer

AMMO uses TCP/IP networking both to connect Android devices to the gateway, and to connect gateway plugins to the gateway core. In the gateway, this networking layer is implemented using ACE (the Adaptive Communication Environment)'s Acceptor/Connector and Reactor frameworks, which abstract away most of the complexity of managing multiple clients and connections, and provide a simple event-driven interface to send and receive data from clients. The data that's actually transmitted over the network is encapsulated inside Google's Protocol Buffers[1], a lightweight, cross-platform data serialization format [1].

Google Protocol Buffers was chosen as the data serialization format for AMMO because it provides simple, efficient serialization of data that isn't dependent on a particular programming language, operating system, or processor architecture. This is important because AMMO needs to communicate between gateway servers, which are coded in C++ and run Linux, and handheld clients, which run the Android operating system. Like the gateway systems, Android is Linux based, but are programmed in Java and typically use ARM processors rather than the x86 processors used in the gateway). Protocol Buffers provides a code generator and language bindings for both C++ and Java. This allows the protocol to be specified in a simple, language-independent text format; Protocol Buffers itself takes care of serializing and deserializing data based on the protocol defined by the developer.

Protocol Buffers also makes the development process of the Gateway and the Android services simpler. Protocol Buffers' code generator takes the language-independent protocol specification and generates a library for message construction and serialization native to the target language (either C++ or Java). This code generator also allows protocols to easily be extended: adding new messages or data elements to the protocol is as simple as adding a few lines to the protocol specification. Also, Protocol Buffers is designed to

[1] http://code.google.com/apis/protocolbuffers/

allow backwards-compatibility between older versions of a protocol and newer versions: it provides a mechanism by which older elements of a protocol can be kept stable, while elements added more recently can be ignored if not present.

The gateway was implemented as two servers. The first, called the Gateway Core, is the endpoint to which all gateway plugins connect. As the name implies, the Gateway Core contains all the core logic of the gateway: the routing logic for data, user authentication logic, and plugin management. The gateway core accepts one (or more, depending on the plugin) connection per connected plugin. All gateway operations are sent over this connection: subscription requests, published data, pull requests and responses, and authentication requests and results. Gateway plugins themselves don't have to be concerned with this network communication, however; the gateway provides an API (called LibGatewayConnector) which abstracts all the details of the network interface away from the programmer and provides a simple C++ API to communicate with the gateway, as though the plugin were running in the same process as the Gateway Core itself. This API is described in section II.4.

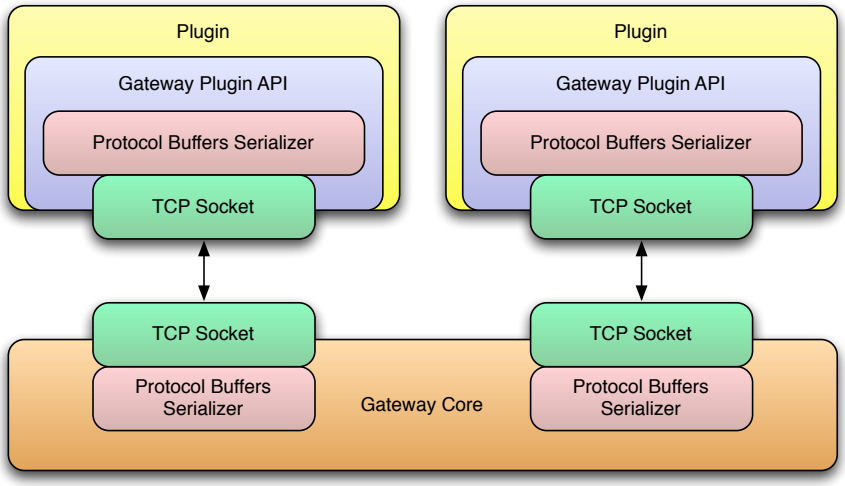

Figure II.2: The AMMO Gateway and Plugins

The second server in the gateway manages communication with Android devices. This

server, called the Android Gateway Plugin, is implemented as a gateway plugin using the LibGatewayConnector API. It accepts a connection from each Android device using AMMO services, and acts as a proxy for the device with the Gateway Core, forwarding data and requests back and forth between the core and the device. The Android plugin's protocol and set of supported operations are very similar to the Gateway Core itself, but it was implemented as a plugin (rather than allowing devices to connect to the Gateway Core directly) for security and stability reasons, and to allow the Android protocol to be expanded later with features like encryption, data prioritization, and compression which may not be necessary or desirable for the inter-process communication between the Gateway Core and its plugins.

II.4 Gateway Plugin API

As mentioned above, gateway plugins communicate with the Gateway Core via a library called LibGatewayConnector. This is an object-oriented C++ API, and hides all the network communication required to communicate with the Gateway Core from the plugin developer. The LibGatewayConnector API is shown in the class diagram in figure II.3, and described below.

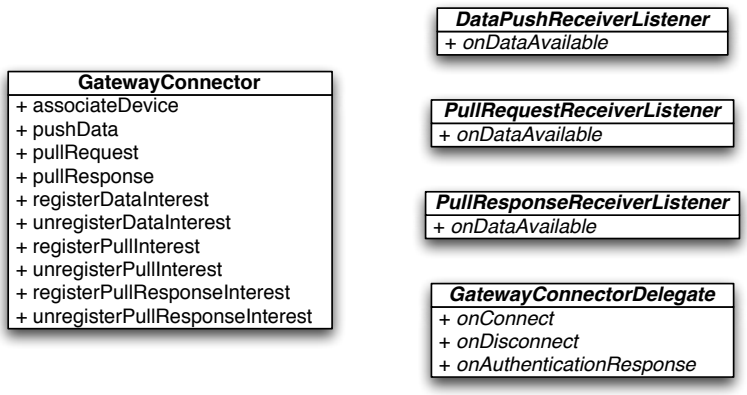

Figure II.3: LibGatewayConnector Class Diagram

On initialization, each plugin creates a `GatewayConnector` instance. When constructing its `GatewayConnector` object, the plugin provides it with an instance of a `GatewayConnectorDelegate` subclass that it has implemented; this is a class of callback methods used for lifecycle management, called upon connection to the gateway, upon disconnection from the gateway, and upon completion of user authentication. The plugin then can do a number of things with the `GatewayConnector`: it can publish data, subscribe to data, register as a pull request handler, or perform any of the other operations described in section II.2.

A plugin publishes data using the `pushData` method. This method accepts a MIME type corresponding to the data type, a URI providing a unique identifier for the piece of data, and, of course, the data itself (which can be any arbitrary binary data; the gateway doesn't restrict the form of the data transferred with it). That data is then transmitted to the Gateway Core, which distributes it to all the plugins and devices which have subscribed to that particular type. Pull requests happen in a similar way– the plugin calls the `pullRequest` method, which accepts the MIME type of data that's being requested, the ID of the plugin requesting the data (used for routing), a query, and an optional projection, which specifies a plugin-specific transformation to be done on the data. The gateway then routes the request to all the plugins which claim to be able to handle requests of that type, and the plugins return all the data which matches the query given in the pull request. Prior to sending a pull request, a plugin must also call `registerPullResponseInterest`, which registers a `PullResponseReceiverListener` which will be called when data is received from the pull request.

A plugin subscribes to data of a particular topic with the `registerDataInterest` method. This method accepts an instance of a `DataPushRecieverListener` subclass, which contains an `onDataAvailable` callback method which is called whenever data of the registered MIME type is received. Likewise, a plugin registers to handle pull

requests of a particular type using the `registerPullInterest` method, which accepts an instance of a `PullRequestReceiverListener` subclass. This class contains an `onDataAvailable` callback method that is called whenever a pull request of the registered type is received, and the plugin responds to the pull request by calling `pullResponse` on the GatewayConnector for each piece of data that it wants to return.

Table II.1: Summary of Gateway API Methods

Method	
pushData	Publishes a piece of data
registerDataInterest	Subscribes to data of a particular type
pullRequest	Sends a pull request to registered pull request handlers
registerPullResponseInterest	Registers a callback method to be called when data is received from a pull request
registerPullInterest	Registers a plugin to handle pull requests of a specific type
pullResponse	Called by a pull request handler to send data in response to a pull request

II.5 Disconnected Operation

One critical feature of AMMO is the ability to support disconnected operation: where a device spends some amount of time without network connectivity and later reconnects, sends any new data it has collected to the server, and receives any new content that was created while it was offline. In our design, this synchronization is handled by a gateway plugin. A plugin subscribes to all data of the types that need to be stored for later retrieval, and registers itself as a pull request handler. When a device reconnects to the network, it issues a pull request for items of that type which were sent while it was offline, and the storage plugin retrieves those items and sends them to the device.

A simple, generic data storage plugin which can handle data of any type and perform

simple queries based on fields such as items' timestamps is provided with AMMO. If developers have more specialized needs, such as querying based on information contained in a message's payload, a storage plugin specialized for that type should be created.

II.6 Functional Test Suite

Along with the core C++ components that make up the gateway, a Python-based testing library was constructed. This library was designed to allow easy construction of functional, stability, and scalability tests of the gateway. The test suite allows the functionality of the gateway to be verified by itself, without using an application on an Android device. This makes it easier to identify which component has introduced an issue, because issues in the Android API don't appear in the Python test suite.

The Python test suite was constructed using the Twisted networking framework[2]. It was designed to emulate the behavior of an Android device: it connects to the Android Gateway plugin, and communicates using the same protocol as a real Android device. The Python test driver supports all of the same functionality as an Android device, including subscribing to and publishing data and executing pull requests.

A number of test drivers have been constructed using our Python test framework. These include tests of the gateway itself, including tests which verify the correct functionality of the gateway's core operations (such as publishing and subscribing to data) and performance tests, such as latency measurements. The test framework has also been used to construct tests for gateway plugins, to verify correct behavior for components such as the data store plugin.

II.7 Android Framework

Although it is not the focus of this paper, a brief discussion of the AMMO components running on the Android mobile devices is worthwhile for understanding AMMO as a whole.

[2] http://twistedmatrix.com/trac

The AMMO Android framework is built around the concept of content providers, which, in Android, provide a way for all applications to store and retrieve data. Content providers are typically backed by a database, and support SQL-style queries as well as automatic notifications when data is added or changed.

In AMMO, an application specifies a content provider that will be used by the AMMO distributor, or generates a content provider appropriate for use with AMMO using a code generator provided with the framework. Data from that content provider is then published, either automatically when added to the content provider or on demand. Subscriptions are also handled by these content providers: data received as part of a subscription is automatically added to the appropriate content provider for its type, and applications monitoring that content provider are notified that new data is available. This provides a straightforward way to produce data-driven networked applications that are well-integrated both with Android and with the AMMO network.

CHAPTER III

IMPLEMENTATION

As mentioned earlier, all of the core parts of the gateway were implemented in C++ using the ACE programming toolkit. C++ was chosen as the implementation language for the gateway because it allows the gateway to be highly performant with a minimal memory footprint, which is important as the gateway may need to operate in a resource-constrained environment. However, writing portable, maintainable code in C++ can be difficult. For this reason, the ACE framework was used to construct the gateway. At its lowest level, ACE provides platform-independent wrapper functions for operating system features such as networking, threading, and file access [7]. Built on top of this low-level platform-independence layer are object-oriented interfaces for much of this same functionality. This allows system functionality such as networking or threads to be well-integrated into C++ code; traditionally, network socket or threading code would be written using low-level C APIs such as the BSD socket API or the pthreads API.

On top of these object-oriented wrapper classes, ACE provides implementations of a number of high-level design patterns. These include patterns such as the Reactor pattern [8] and the Acceptor-Connector pattern [9], both of which are used heavily in the gateway. This library of patterns reduces the amount of boilerplate code that must be written to create a functional application, allowing more time to be spent developing logic unique to the gateway.

III.1 Network service implementation

All the components of the gateway, including the Gateway Core, Android Gateway Plugin, and Gateway plugin API, use very similar networking code, using ACE's Acceptor-Connector framework on top of the ACE Reactor. These provide an event-driven interface

to network communication, where the Reactor handles waiting for events such as incoming data and dispatching them to their appropriate event handlers [8] [9].

The Gateway Core and Android Plugin both implement network servers, and therefore use the ACE Acceptor, the part of the ACE Acceptor-Connector framework which passively accepts connections from clients using the Connector pattern [9]. As used in the gateway, the Acceptor creates a TCP listening socket and waits for connections. When a connection is established, the Acceptor framework creates a new event handler object for that connection and registers it with the reactor so it can be used to handle events from the connection it manages. Likewise, the Gateway plugin API acts as a network client (connecting to the Gateway Core). Each instance of the `GatewayConnector` class creates an ACE Connector, which attempts to connect to the Gateway Core, and, on connection, creates an event handler object for the new connection.

This event handler object, or service handler, as it is called within the ACE framework, contains all the logic required to communicate with a client. It contains all the state required to manage a single connection, including message queues, the network socket of the connected client, and other miscellaneous data needed by the gateway to manage that connection. This structure, where each active connection to the gateway has its own service handler, provides a logical way to keep data for each connection separate while making it simple to access a specific connection where it is needed.

Each component of the gateway (the Gateway Core, Android Plugin, and gateway API) provides a service handler class which inherits from the `ACE_Svc_Handler` class. This class defines several methods, including `open`, `handle_input`, `handle_output`, and `handle_close`, which are called by the Reactor when the appropriate event occurs for the connection the service handler is associated with:

- `open` is called when a connection is established. This method sets up the initial state of the service handler and performs any other actions which must be done when a connection is initiated.

- `handle_input` is called when data is available to read on a socket. In each gateway component, this is implemented as a state machine: it first reads a short header containing a message size and checksum, then reads the number of bytes specified by the header. Due to the nature of network communications, these reads may be spread out across multiple `handle_input` calls. Once an entire message has been read, the data is passed off to a `processData` method, which validates the checksum from the header, deserializes the Protocol Buffers message contained within, and performs some action based on the contents of the message. For the Gateway Core, this `processData` method will perform operations on the Gateway Core as described in section III.2.

- `handle_output` is called when the application may write to a socket. In the gateway, this method gets the next message to be sent off a queue, serializes that Protocol Buffers message, computes the checksum and size of that message, then sends the header containing size and checksum followed by the message itself.

The service handler classes also include state information pertaining to the connection they are associated with. This state information includes the information required to manage the `handle_input` and `handle_output` methods: current state machine state, buffers of received data, and the number of bytes sent or received. As mentioned above, this also includes a queue of messages to be sent over the socket. This queue allows the serialization and sending of a message to be decoupled from the point where a message is generated. If messages are sent as they are generated, the gateway may have to block inside an event handler, which impairs performance. Instead, messages are queued and sent when the `handle_output` event is dispatched indicating that the application may send data without blocking.

III.2 Gateway Core implementation

The gateway core itself is structured as a singleton class which is shared between all active service handlers. This class maintains the routing tables used in the gateway and manages the actual distribution of data between plugin connections.

Routing information for messages is maintained as a number of maps. Subscriptions and pull request handlers are stored as C++ STL multimaps (where one key can map to more than one value), with the subscription type name as the keys of the maps, and pointers to the service handler representing the connection which made the subscription or pull request as the values of the maps. A multimap was used for these tables as more than one plugin may subscribe to a given type, and the STL multimap specifically provides a reasonably efficient, portable implementation of this data structure.

The routing information for pull responses is stored as a C++ STL map, with the name of the plugin which made the original request as the keys of the map, and a pointer to the service handler representing that plugin as the values of the map. Unlike subscriptions and pull request handlers, there should not be more than one plugin with the same name connected to the gateway at one time. Therefore, a map is used instead of a multimap to enforce this requirement.

The process through which a message is dispatched by the gateway can be best illustrated with an example, shown in figure III.1. Consider a push message sent from a gateway plugin. Once the entire message has been received by the service handler for that plugin connection, the message's checksum is verified, and the Protocol Buffers deserializer is called to deserialize the message. The service handler's `processData` method is then called, which identifies the message as a Push message, and calls the `pushData` method on the singleton `GatewayCore` object. The GatewayCore then gets all the service handlers from the subscription map which match the type of the message being pushed, and

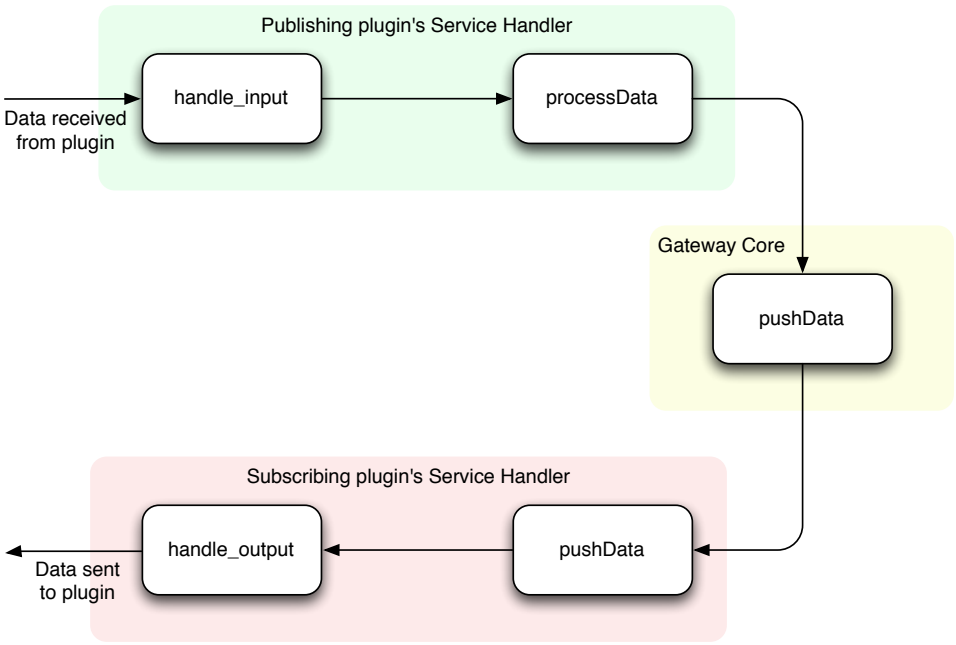

Figure III.1: Gateway Message Flow

calls the `pushData` method on each of those service handlers. This method creates a Protocol Buffers message containing the data to be pushed, serializes it, and adds it to the service handler's outgoing message queue. From there, the message will be sent once all messages ahead of it in the queue have been sent and the service handler's `handle_output` method has been called.

One important implementation decision made while implementing the Gateway Core was to keep the Gateway Core single threaded, in a single process. This means that all network I/O and all event handling happens serially, rather than allowing network I/O or event handlers run simultaneously. This was done primarily for simplicity of implementation. Making the Gateway Core multithreaded would add complexity, as the data structures used by all service handers, such as the subscription maps, would need to be synchronized to ensure consistent operation between threads. Although this is not infeasible, the single threaded implementation is simpler, which simplified development and debugging.

III.3 Android Plugin implementation

The Android Plugin, the gateway plugin which manages connections between Android mobile devices and the gateway, is considerably simpler than the Gateway Core. The Android plugin does not contain much logic of its own; it simply receives requests and forwards them on to the Gateway Core for processing. The Android plugin is a consumer of the LibGatewayConnector gateway plugin API described in section II.4, and acts as a network server to receive connections from devices as described in section III.1. When the Android plugin receives a message and its `processData` method is called, it simply calls the appropriate method of the gateway plugin API to forward that data to the Gateway Core. The Android plugin also registers subscription callbacks on behalf of connected devices; when published data is received from the Gateway Core, the Android plugin will send that data to the appropriate connected device.

Unlike the Gateway Core, the Android plugin is multithreaded. While network communication all happens in a single thread, each connection has its own thread where messages are processed. This decouples message processing completely from network communication; message queues are used to communicate between two levels. This decoupling ensures that slow network communications to or from one device can never prevent message processing from happening for another device.

CHAPTER IV

RESULTS

To evaluate the performance of the gateway, the Python test driver framework described in section II.6 was used to measure the end-to-end latency of messages sent to the gateway. This test suite was composed of three components: a producer script, which generates messages containing a timestamp and a user-specified amount of random data and sends them to the Android gateway plugin; a consumer script, which subscribes to the messages generated by the producer script and computes the total time taken to receive the message based on the timestamps; and a driver script which creates a producer and a user-specified number of consumers and allows them to run for a set amount of time.

The Python testing framework was chosen for this test rather than using the AMMO API on the Android devices to eliminate the added variability introduced by the Android API. In the Android API, data must flow through a number of components before it is serialized and sent over the network to the Android plugin. Also, Android devices are processor, memory, and bandwidth constrained. Since the focus of this test is on the Gateway's performance, the Android components were simulated by the Python test framework to eliminate as many extraneous variables as possible.

All tests were conducted using two computers. One machine, running 32-bit Red Hat Enterprise Linux 6 (the standard deployment environment for the Gateway) was used as a dedicated gateway machine. The second machine ran 64-bit Ubuntu Linux 11.04, and was dedicated to the Python-based producer and consumers. The computers were connected via gigabit ethernet, to minimize the impact of network bandwidth on the latency results.

Two main tests were conducted. The first test was designed to determine the impact of multiple consumers on message distribution. A single producer was created, and the test was repeated for 1, 5, 10, 20, 40, and 60 consumers, with message payload sizes (in

addition to the included timestamps and Protocol Buffers overhead) of 0, 20, and 1000 bytes. A message rate of approximately two messages per second was used, and the test was allowed to run for 500 seconds, for a total of approximately 1000 messages sent per test. The latency from the producer to each consumer was measured, and averaged across all consumers.

The results from this test are shown in figure IV.1 . These graphs show what appears to be a roughly logarithmic scaling of message latency with respect to number of consumers. This is a desirable property, because it means that additional consumers impact overall latency less as the number of consumers increases.

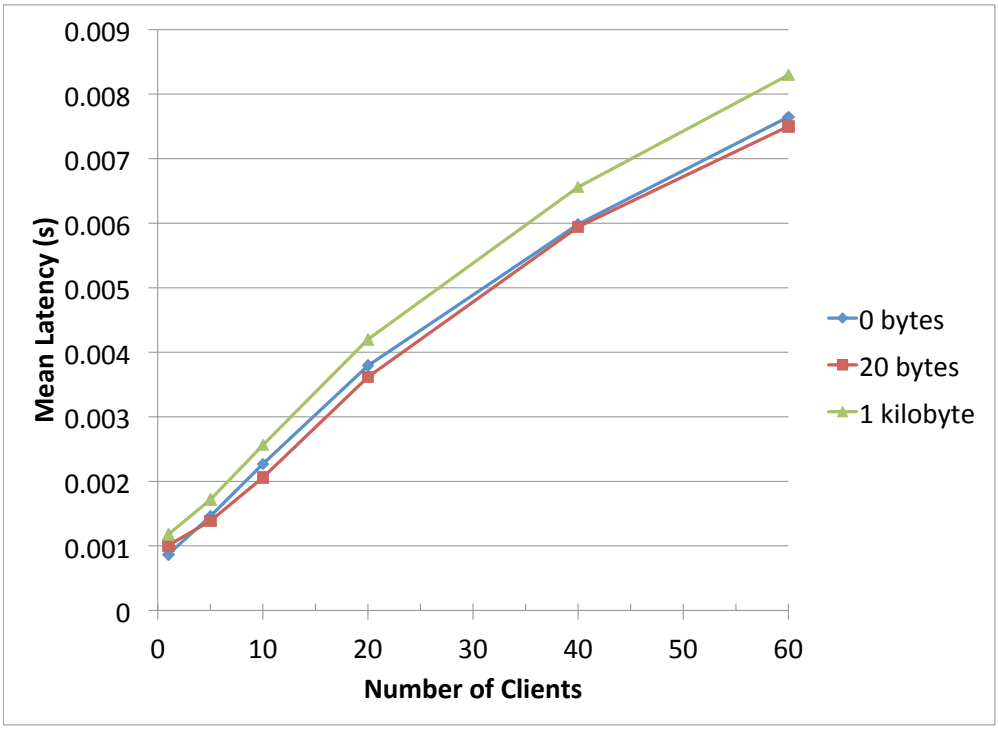

Figure IV.1: Effect of number of clients on latency

The second test was designed to determine the impact of message size on message

distribution. A single producer and 10 consumers were created for each run of the test. The tested message payload sizes were 0 bytes, 20 B, 500 B, 1 KB, 10 KB, and 50 KB, representing different possible message sizes for textual and multimedia content. As with the first test, a message rate of approximately two messages per second was used, and the test was allowed to run for 500 seconds, giving a total of approximately 1000 messages sent per test.

The results from this test are shown in figure IV.2 . These graphs show an approximately linear scaling of message latency with respect to message size. This is as expected, as message transfer times over a network as well as data deserialization and copy times scale linearly with respect to size.

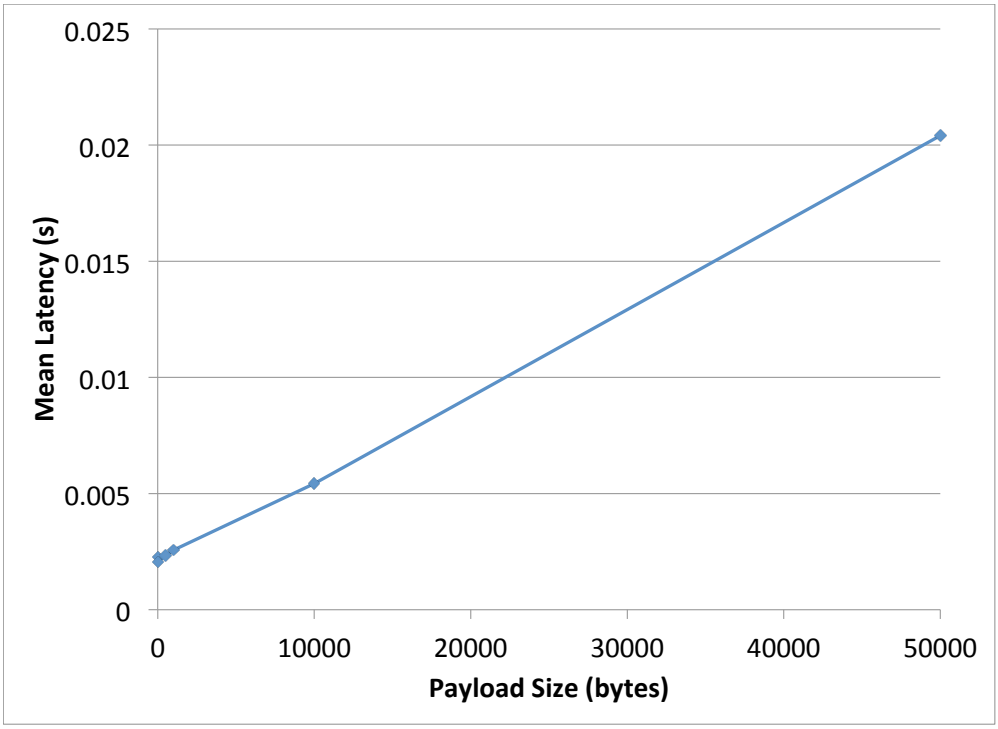

Figure IV.2: Effect of message size on latency

CHAPTER V

RELATED WORK

The pioneering example of the publish/subscribe communication model is the Object Management Group, or OMG, Data Distribution Service (DDS)[1]. DDS is a specification for publish/subscribe data distribution systems, and provides a common, platform-independent interface that defines the data distribution service [6]. The DDS specification itself provides a UML model which specifies the interface to a data distribution service, which then can be mapped onto a number of platforms and programming languages. A number of implementations of this standard exist, including OpenSplice DDS, OpenDDS, and RTI DDS.

DDS defines what they call a "Data-Centric Publish-Subscribe" interface, which provides participants in the system (publishers or subscribers) with an efficient, typed interface to read and write data. The DDS middleware then distributes the data such that each subscriber is able to access the current values for any piece of data [6]. To give the developer control over this distribution process, the DDS specification also defines a set of quality of service (QoS) policies. These policies can be attached to any publisher or subscriber, and set constraints on the data distribution process, such as the allowable latency of data distribution or the rate at which data must be updated.

Like DDS, the AMMO gateway provides a publish-subscribe paradigm for distributing data. AMMO is not, however, an implementation of the DDS specification. AMMO uses a much simpler API for data publishing and subscribing, both on the gateway and on Android devices: this allows AMMO to operate in mobile environments where network bandwidth is limited and low power consumption is essential. AMMO also does not currently offer an

[1] http://www.omgwiki.org/dds/

equivalent to DDS's quality of service policies, although support for end-to-end quality of service guarantees is planned for future work.

Many other attempts have been made to bring this publish-subscribe paradigm to mobile devices. It is worth mentioning a few of these to provide context and comparisons to AMMO:

STEAM [5] is a publish-subscribe system designed specifically to operate over 802.11 WiFi ad hoc networks. Therefore, unlike AMMO, STEAM was designed to operate without a gateway; it operates as a fully distributed system. STEAM also provides a richer but more complicated method of filtering and distributing data: rather than filtering by topic alone as done in the AMMO gateway, STEAM filters objects based on topic, proximity, and content. [5]

Pronto [2] is a system providing both publish-subscribe and point-to-point communication, based on the Java Messaging Service standard. Pronto can operate in two modes: a centralized mode where a gateway, much like the AMMO gateway, distributes data, and a decentralized mode where devices share data amongst themselves. Also, in addition to gateways residing on remote servers, Pronto clients can communicate with gateways residing on the devices themselves, alongside the clients. This is distinct from the current iteration of AMMO, where gateways can't reside on the devices. Pronto also puts a heavy emphasis on what they call "SmartCaching" in the gateway. This is directly analogous to what the gateway provides with its data store plugin, which ensures data is delivered even under intermittent connectivity. [2]

Pervaho [3] is also a publish-subscribe system, but distinguishes itself from AMMO and the other systems mentioned by focusing specifically on location-based publish-subscribe, where devices receive data related to locations which they are in physical proximity to. Like AMMO, the current implementation of Pervaho uses a centralized gateway to receive location updates and distribute data, although the authors mention that they are interested in modifying Pervaho to use a decentralized model.

These different approaches to publish-subscribe communication in a mobile environment, as well as a couple others used in some systems, are summarized in [4]:

Centralized: A central broker stores all subscriptions in the system. This is the approach currently implemented by the AMMO gateway.

Centralized with Quenching: The central broker gives each event source a filter based on the union of all active subscriptions, so they don't send objects which don't match any active subscriptions to the central broker.

Distributed broadcast: Uses a number of event brokers (rather than a single centralized one), with each broker responsible for a portion of all subscriptions. An event source connects to one of these brokers, which then broadcasts the event to all the other brokers in the system.

Distributed multicast: Events are selectively forwarded between brokers depending on the active subscriptions in the system.

Replication: Each user's subscriptions are monitored by multiple brokers at the same time, making it less likely that a user will miss events. This makes it more difficult to ensure that duplicates are not delivered, that events arrive in order, and that all events are actually received by all subscribers.

The AMMO gateway currently implements the centralized model as described in this paper: the AMMO gateway is the centralized point where all subscriptions and messages must be sent. However, AMMO extends the pure publish-subscribe approach given in this paper and provides a means by which devices can explicitly request (or "pull") data from data sources. Also, [4] does not address the issue of integration of existing services and data sources: AMMO provides a framework for this.

CHAPTER VI

CONCLUSION AND FUTURE WORK

Overall, the AMMO gateway provides an effective back-end system to support the AMMO middleware running on mobile devices. It provides a centralized point of communication to allow the device middleware to optimize bandwidth utilization, quality of service, and power consumption, and provides an integration point for third-party services to interact with applications running on an AMMO-enabled network. It makes efficient use of bandwidth by using Google's Protocol Buffers as a network transport protocol, and it provides a central point for the implementation of security and quality of service guarantees across multiple applications and services.

As currently implemented, the AMMO gateway contains the basic, core functionality required for data dissemination and distribution. However, it is a relatively simple implementation; further work is required to make it a powerful, flexible environment for data distribution.

One area which is being emphasized is support for multiple federated gateways. These gateways could be placed in physically separate locations, and provide a bridge for handhelds and services operating in different areas to communicate between among each other. This is similar to the distributed broadcast approach in [4]. As shown in figure VI.1, many gateways (G) could be connected via possibly redundant network links, with different devices (D) and services (S) connected to each gateway. Data published by a device or service connected to one gateway could then be distributed to all the devices and services connected to all the other gateways. The route that data and subscriptions take through the network should be determined by a policy determined by the system administrator; this

should allow the network to be configured to use the best possible route given network conditions, and to fail gracefully by falling back to other, redundant links when a connection fails.

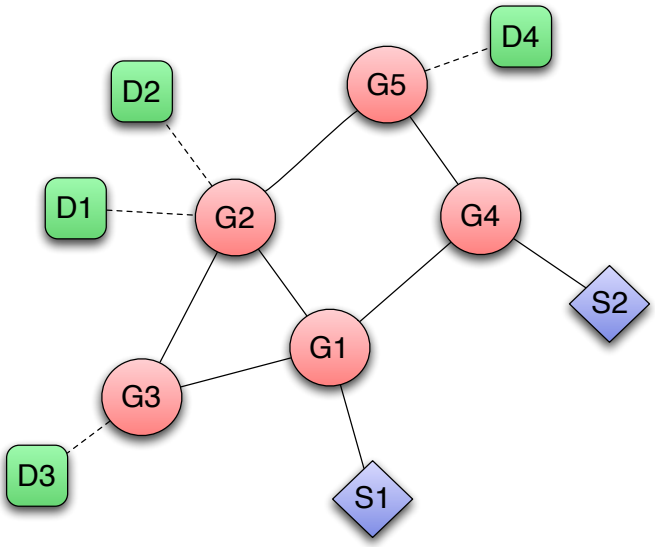

Figure VI.1: Network of Gateways

Another important area of future work is support for quality of service guarantees. While we do not want to support the wide range of policies specified by the DDS specification, some control over message delivery is necessary is necessary in the environments AMMO will be used in. For example, it would be very undesirable if a request for a medical evacuation were delayed because another user was in a video chat. Some policies we may implement are message priorities, multiple levels of delivery confirmation, and durability policies which control the lifetime and validity of objects.

A third possible area of future work would extend subscriptions to allow filtering based on the data contained within each object. Currently, subscriptions are based on message type; a device or plugin always receives all published data of the types it is subscribed to. However, there may be situations where a user isn't interested in all the data of a type; for example, they may only be interested in reports created at nearby locations. This

isn't possible with our current implementation, because data is encoded in an application-specific format which the gateway may not understand. To achieve this, messages would need to include metadata in some common, gateway-readable format, which the gateway can then process and filter before forwarding messages on to devices.

REFERENCES

[1] Developer guide - protocol buffers, October 2011. URL http://code.google.com/apis/protocolbuffers/docs/overview.html.

[2] Jean Bacon Eiko Yoneki. Pronto: Mobilegateway with publish-subscribe paradigm over wireless network. Technical Report 559, University of Cambridge Computer Laboratory, 2003. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.14.8927&rep=rep1&type=pdf.

[3] Patrick Th. Eugster, Beno Garbinato, and Adrian Holzer. Location-based publish/subscribe. *Network Computing and Applications, IEEE International Symposium on*, 0: 279–282, 2005. doi: http://doi.ieeecomputersociety.org/10.1109/NCA.2005.29.

[4] Yongqiang Huang and Hector Garcia-Molina. Publish/subscribe in a mobile environment. *Wirel. Netw.*, 10:643–652, November 2004. ISSN 1022-0038. URL http://dx.doi.org/10.1023/B:WINE.0000044025.64654.65.

[5] R. Meier and V. Cahill. Steam: event-based middleware for wireless ad hoc networks. In *Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on*, pages 639 – 644, 2002. doi: 10.1109/ICDCSW.2002.1030841.

[6] Gerardo Pardo-Castellotte. OMG data-distribution service: Architectural overview. White paper, Real-Time Innovations, Inc., April 2005. URL http://www.omgwiki.org/dds/sites/default/files/DDS_Architectural_Overview.pdf.

[7] Douglas C. Schmidt. The adaptive communication environment: Object-oriented network programming components for developing client/server applications. In *12th Sun Users Group Conference*, June 1994. URL http://www.cs.wustl.edu/~schmidt/PDF/SUG-94.pdf.

[8] Douglas C. Schmidt. Reactor: An object behavioral pattern for concurrent event demultiplexing and event handler dispatching. In *Proceedings of the 1st Pattern Languages of Programs Conference*, August 1994. URL http://www.cs.wustl.edu/~schmidt/PDF/Reactor.pdf.

[9] Douglas C. Schmidt. Acceptor and connector: A family of object creational patterns for initializing communication services. In *Proceedings of the European Pattern Language of Programs Conference*, pages 10–14, July 1996. URL http://www.cs.wustl.edu/~schmidt/PDF/EuroPLoP.pdf.