Systems for the Interconnection of Android Applications and Automobiles via the OpenXC Framework

by
Arkady Blyakher
S.B. EECS, M.I.T., 2012

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2013

Author ....
Department of Electrical Engineering and Computer Science
February 1, 2013

Certified by ....
James R. Glass
Senior Research Scientist
Thesis Supervisor

Certified by ....
Scott Cyphers
Research Scientist
Thesis Supervisor

Accepted by ....
Prof. Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

Systems for the Interconnection of Android Applications and Automobiles via the OpenXC Framework

by
Arkady Blyakher
S.B. EECS, M.I.T., 2012

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

Abstract

The topic of this thesis is the design and construction of a set of applications that use information from a vehicle to provide a better user experience. We introduce two Android applications that make use of the open-source OpenXC library to gauge driver awareness and to provide a user interface via steering wheel controls. The first of these applications deals with human-to-human interactions in the form of a messaging client. Our goal for this application is to provide a method of determining when to notify drivers of new messages by using vehicle data to gauge driver awareness. The second application deals with human-to-machine interactions in the form of a point-of-interest browser. Our goal for this second application is to use steering wheel controls in place of the touch screen traditionally associated with Android mobile applications. We hope to demonstrate that our versions of these mobile applications, with their focus on preserving driver awareness, provide a viable upgrade to their traditional alternatives.

Thesis Supervisor: James R. Glass
Title: Principal Research Scientist

Thesis Supervisor: Scott Cyphers
Title: Research Scientist

Acknowledgements

First, I would like to thank Jim Glass and Scott Cyphers, who were my thesis co-advisors. This work would not have been possible without their help and guidance. I greatly appreciate their patience and insight throughout this project.

I would like to thank John Leonard, who first introduced me to the ideas that formed the basis of this work. It is thanks to his efforts that I came in contact with OpenXC, and was able to obtain a test vehicle.

I would like to thank Andrew Patrikalakis for all of his work. He helped to lay the hardware foundations used by this project. His insight and technical expertise were crucial, and he helped me through several roadblocks over the course of this project.

I would like to thank the people at Ford and on the OpenXC project who made this work possible.

Most importantly, I would like to thank my family for always supporting me, for the sacrifices they made for me, and for inspiring me. None of what I achieved would have been possible without them.

Contents

Figures and Tables

1 Introduction

1.1 Adoption of Mobile Technology

Our current lifestyles have shown a slow shift into a more technology-centric realm. We have smartphones, tablet PCs, and the "cloud." A Nielsen survey taken January 2012 found that 66% of Americans between the ages of 24 and 35 own a smartphone[1]. Similarly, the market analyst firm Gartner predicted that tablet PC sales will reach 119 million units by the end of 2012. This is an increase of 98% over the 60 million units sold in 2011.[2] These figures exemplify the current trend towards mobile computing. As consumers, we expect to have our computers manage our lives, entertain us, and provide us with a social outlet. Yet, we do not expect to be stuck behind a desk to take advantage of all of these features. There are currently thousands of free Wifi hotspots across the US, not to mention the fee-based services provided by all of the major telephone carriers. With the market for mobile computing growing quickly, it is inevitable that more of our everyday tasks incorporate the ability to interact with these devices.

1.2 Mobile Technology in Automobiles

One major, as yet untapped, resource comes in the form of the automobile. The average American spends 18.5 hours in the car each week, as either a driver or a passenger[3]. While this might not seem like a lot at first, the chunks quickly add up. At 18.5 hours a week out of our waking hours, we are spending nearly 1.3 months out of the year in our cars. At the same time, we are severely limited by what we can do while driving. Many current mobile applications do not take driving into account when providing a user interface. Driver distractions are a big hazard to everyone on the road, with many states adopting more stringent guidelines on what drivers can and cannot do while behind the wheel. With these changes, phones and other mobile devices will need to change as well, to properly reflect the different circumstances that an individual faces while in a car.

1.3 Age Trends Between Automobiles and Mobile Devices

The average age of a vehicle on U.S. roads is over 11 years. This figure comes from the findings of the research firm Polk, which tracks and releases data for the automotive sector. This figure continues the gradual trend in the increase of the average age of both passenger and commercial vehicles. For comparison, the average age of an automobile on U.S. roads was 8.4 years in 1995, and 10.4 years as recently as 2009.[4]

The aforementioned aging trend for U.S. automobiles has not been mirrored in the mobile computing sector. In contrast, we have seen computer prices fall significantly within the past decade, along with steep depreciation of existing computer hardware. This phenomenon, combined with the rapid pace of innovation within the mobile computing industry, has contributed to a low mean-time-between-replacements rate for mobile devices. Consumers are replacing laptop computers after 4 years and 4 months, on average. Similarly, cellphones are upgraded, on average, more recently than every two years.[5]

1.4 Thesis Outline

This work is divided into several chapters. The first serves as an outline of the current state of U.S. automobiles, and their relationship with mobile devices. We hope to highlight the predicaments faced by drivers currently using mobile devices in their vehicles. In this context, we start our second chapter with a description of the purpose of our work: to develop a suite of OpenXC-enabled applications that allow us to create driver-centric experiences on mobile devices. We present a target scenario that we want our suite to facilitate. Our third chapter provides the reader with an overview of OpenXC, an Android library that allows Android developers to receive data from a variety of Ford vehicles. The third chapter provides a background on OpenXC, and how it is used with Android application. The fourth chapter of this work deals with framework considerations that we faced when deciding how to integrate different OpenXC-enabled applications. We discuss both centralized and decentralized organizational frameworks, as well as the design we settled upon. The fifth chapter covers the OpenXC-enabled applications that we constructed: CarTalk and CityBrowser. The first section of the fifth chapter covers the implementation of CarTalk, as well as the CarTalkServer that it requires for its functionality. We discuss our implementation and design decisions, while giving an overview of how the CarTalk application operates. The second segment of the fifth chapter deals with the CityBrowser application. We start the second segment with an overview of the Asgard City Browser, the non-OpenXC-enabled predecessor to CityBrowser. We go on to detail the specific design challenges that came with adapting the Asgard City Browser for our specific use case, and we describe the ways that we dealt with these challenges. The sixth chapter of this work provides a summary for the work we have achieved, and introduces future lines of investigation.

2 Purpose

2.1 The Idea of Plug-In Hardware

The difference between aging trends for automobiles and mobile devices shows that car electronics are getting progressively older compared to what is available on the consumer market. We see the results of this aging in dated user interface designs and slow processing times. One solution to this problem is to separate the mobile computing hardware from the purchase of the vehicle itself. With plug-in mobile computing systems for automobiles, a new set of design constraints must be implemented. In this work, we examine and deal with these constraints, presenting a working and extensible implementation of a framework for using mobile computers, namely Android devices, in OpenXC-enabled Ford automobiles.

2.2 Project Goal

The goal is to use OpenXC-enabled applications for both human-to-machine and human-to-human interactions. We present two example applications in our design, each offering distinct architectural challenges. First is CarTalk, a messaging client that uses OpenXC measurements to provide for a safer method of communication.

CarTalk uses measurements to make assumptions about the driver's state of awareness and accordingly chooses when to either present or withhold the delivery of new messages. Second is CityBrowser, which offers users the ability to locate points of interest using speech queries. However, this application was not originally designed to be accessible via Android, or to be used in conjunction with OpenXC. This presented a wide array of interesting problems, and led to some major decisions for our overall design.

2.3 Target Scenario

To establish a baseline for our applications, we chose to build a system that would prove effective under the following scenario: a user in an automobile decides to get in touch with a friend for dinner after a long day of work. The user employs our application suite to message her friend, and the two agree on a time but are unsure of a venue. Our user then asks our application suite to provide a suggestion for dinner. She knows that she wants Italian cuisine, maybe even pasta. The application takes her question and returns a list of restaurants in the area. Our user looks over the list, picks a restaurant with a good ranking, and has our application suite forward her decision on to her friend.

2.4 Scenario Constraints

For the established scenario, we wanted to develop a system that would be preferable to existing systems. For example, our system is integrated with the vehicular controls. Similarly, it minimizes the attention that the driver has to divert to perform these tasks. This leaves drivers with fewer distractions, leaving them more able to concentrate on the task of driving.

3 Background

3.1 The OpenXC Framework

To introduce our design scheme, we begin with the OpenXC framework, an integral portion of our implementation. The OpenXC framework is a combination of open source hardware and software that provides a development interface for vehicles. The hardware module reads and translates metrics from a car's internal network, making them available to any Android application that employs the OpenXC library.

OpenXC is an API developed as a joint venture between the Ford Motor Company and Bug Labs. It was created to set a standard for adding aftermarket software and hardware to vehicles[6]. Because every new car is full of computerized and electronic controls, there is growing interest in connecting the output from those systems to third-party applications and to the web. Several tools are available that allow an application or hardware add-on to hook into the driver's interface, but, for the most part, they provide limited capability to hobbyists and developers. OpenXC, on the other hand, was designed to be open source and to give insight into the vehicle itself. At the same time, the OpenXC platform protects the manufacturer from needing to release the vehicle's proprietary implementation details to the public. This latter activity could be risky for a manufacturer as it is possible to reverse-engineer a car given the information in these codes. Worse, an amateur use of these codes in a moving vehicle can be dangerous. Hence, OpenXC strikes a balance between openness of hardware and software within the vehicle, and protection for the manufacturer.

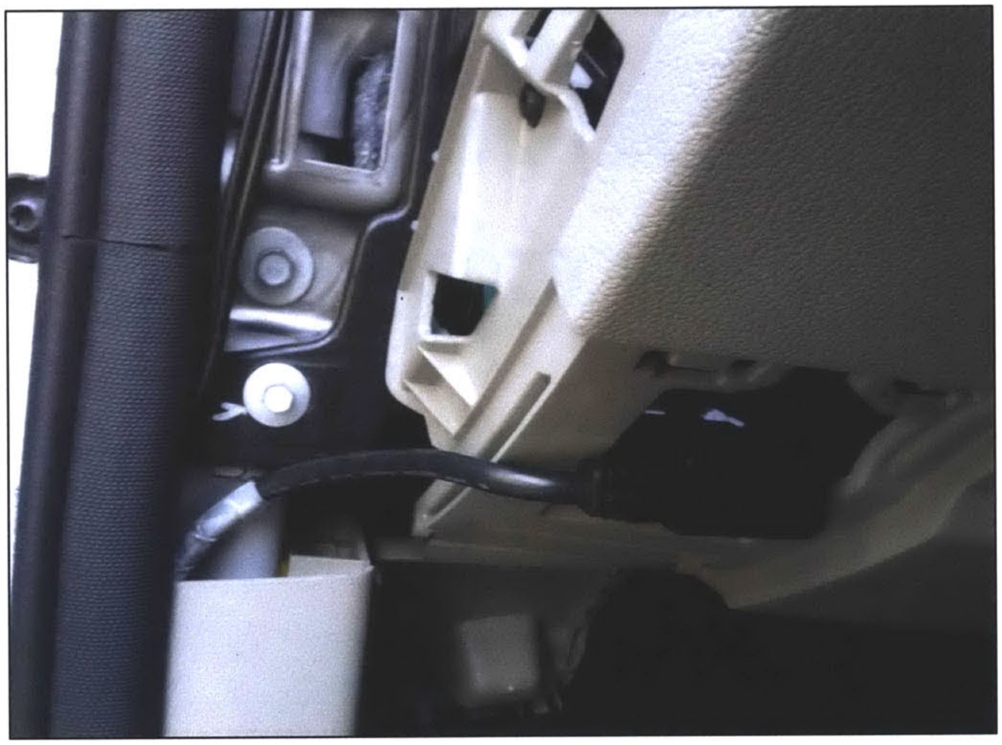
3.2 The CAN Bus AND OBD-II

To get an understanding of how the OpenXC framework operates, we must start with the vehicle itself. Modern vehicles come equipped with a multitude of microcontrollers and sensors. These devices have varying roles, from measuring throttle position and fuel economy to checking for open doors. All of these sensors are on the vehicle's controller area network (CAN) bus. The CAN bus allows the individual electronic control units within the car to broadcast and receive messages. If the bus is free, any node may begin to transmit. In the case where two or more nodes begin sending messages at the same time, the bus performs a priority-based arbitration. Messages with numerically smaller values of IDs are defined as having a higher priority and are transmitted first. Each of the nodes connected to the CAN bus consists of three main components: a host processor, a CAN controller, and a transceiver. The host processer, connected to an array of sensors, is responsible for both deciphering received messages and deciding on which messages to transmit. The CAN controller serves as an intermediary between the host processor and the CAN bus. The controller serializes messages from the host processor before sending them over the bus, as well as deserializing messages arriving over the bus before sending them to the host processor for interpretation. The transceiver acts as a level converter, adapting signal levels between the CAN controller and the bus, as well as protecting the controller from damaging signal levels.[7]

The CAN bus is one the principle components of OBD-II, a vehicle diagnostic standard that has been mandatory for all U.S. cars since 1996. The OBD-II standard, and therefore the messages sent across the CAN bus, are externally accessible via a diagnostic port. The diagnostic port is usually located under the steering wheel, within the driver's footwell.

Figure 1

The OBD-II Port

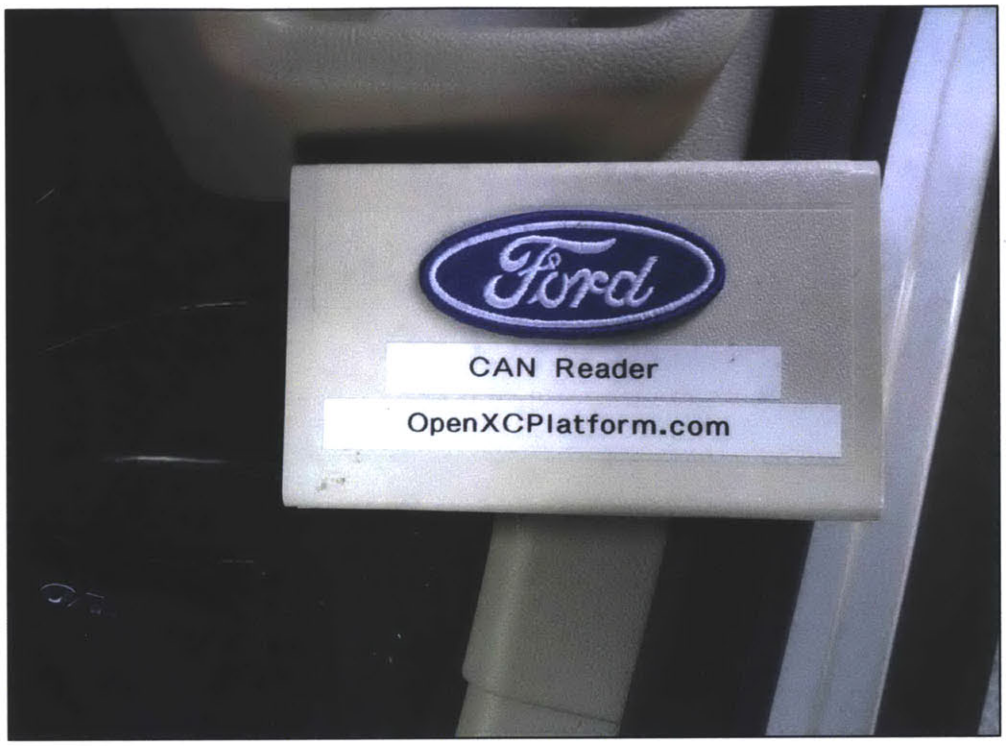


The OBD-II port in the driver's footwell.

The OpenXC hardware connects directly to this diagnostic port. The first of the OpenXC architecture's two components is the CAN translator, which connects directly into the OBD-II port. The CAN translator reads and translates CAN messages, putting them into a standardized format.

Figure 2

The CAN Translator



The OpenXC CAN Translator used with our test vehicle. A list of all supported Ford vehicles is available at http://openxcplatform.com/vehicle-interface/index.html.

Figure 3

Standardized Format for Single-Valued and Event Messages for OpenXC

Single Valued
The expected format of a single valued message is:

```
{"name": "steering_wheel_angle", "value": 45}
```

Event
The expected format of an event message is:

```
{"name": "button_event", "value": "up", "event": "pressed"}
```

* Taken from the OpenXC Vehicle Interface documentation, available at openxcplatform.com/vehicle-interface/output-format.html

Figure 4

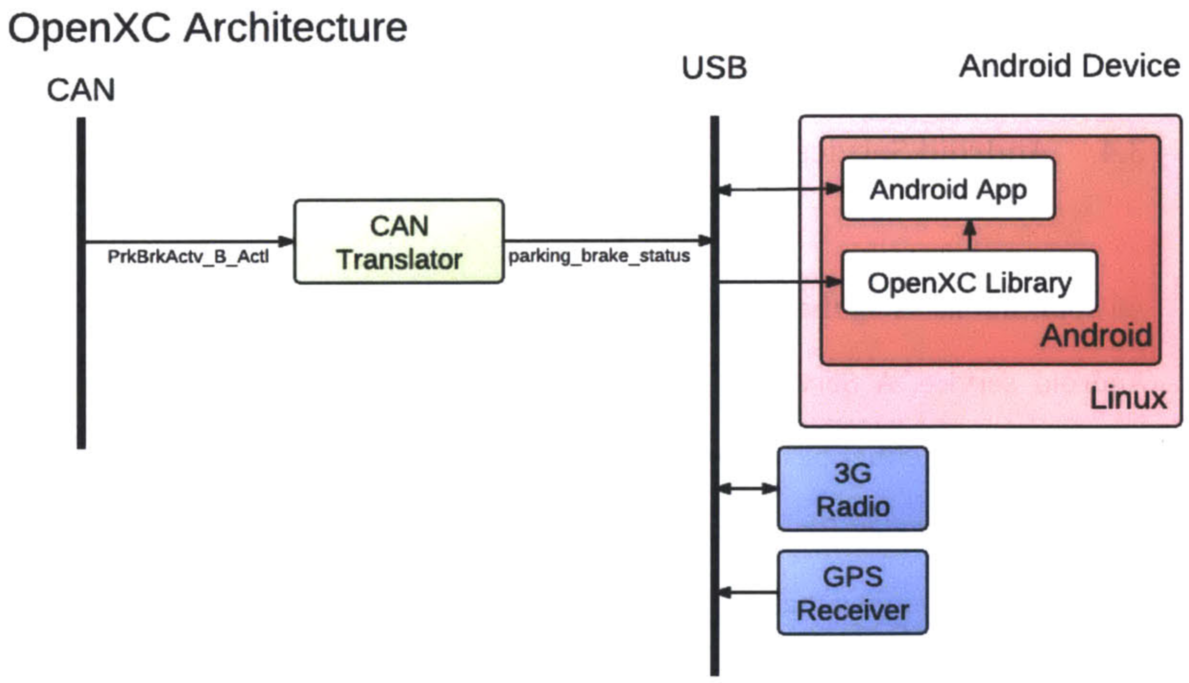
Measurements Supported by OpenXC

Measurement	**Range**	**Units**
steering_wheel_angle	-600 to +600	Degrees
torque_at_transmission	-500 to 1500	Nm
engine_speed	0 to 16382	RPM
vehicle_speed	0 to 655	Km/h
accelerator_pedal_position	0 to 100	%
parking_brake_status	True == brake engaged	Boolean
brake_pedal_status	True == pedal pressed	Boolean
transmission_gear_position	First, Second, Third, Fourth, Fifth, Sixth, Seventh, Eighth, Reverse, Neutral	Gear State
odometer	0 to 16777214	Km
fine_odometer_since_restart	0 to 4294967295.0	Km
ignition_status	Off, Accessory, Run, Start	Ignition State
fuel_level	0 to 100	%
fuel_consumed_since_restart	0 to 4294967295.0	L
door status	Driver, Passenger, Rear Left, Rear Right	Door Value
	True == ajar	Door Event
headlamp_status	True == on	Boolean
high_beam_status	True == on	Boolean
windshield_wiper_status	True == on	Boolean
latitude	-89.0 to 89.0	Degrees
longitude	-179.0 to 179.0	Degrees
button_event	Left, Right, Up, Down, Ok	Button Value
	Idle, Pressed, Released, Held_Short, Held_Long, Stuck	Button Event

* Data for the preceding table has been obtained from the OpenXC Vehicle Interface documentation, available at openxcplatform.com/vehicle-interface/output-format.html

The translated messages can then be sent via Bluetooth or USB to an Android Host Device, the second part of the OpenXC architecture. The host device uses an Android Java library to enable applications to receive the translated messages.

Figure 5



* Diagram from openxcplatform.com/getting-started/index.html

3.3 Device Overview

For our system, our main test device was a Samsung Galaxy Nexus. This phone provides many of the flagship Android features, and we targeted our applications to run on Android 4.0, or Ice Cream Sandwich. However, our application suite has also been tested on devices running Android 3.2 (Honeycomb). Our main requirement was that the devices support USB Hosting, which is required by OpenXC. When an Android device is in USB Host mode, it acts as the USB host, powers the bus, and enumerates connected USB devices. USB host mode is only supported on devices using Android 3.1 and higher, which have the appropriate hardware modules.

3.4 Android Services

All applications using the OpenXC architecture receive vehicle measurements via an Android service. A Service, as described in the Android API, is an application component that either performs a long-running operation while not interacting with the user or supplies functionality for other applications to use. There are some important distinctions that we must address in regards to Android services. First, a service normally runs in the same process as its containing application and not in a separate process. Similarly, a service does not by default spawn its own thread to do work. These considerations were important in our framework designs because we did not wish to lose the open-endedness with which an OpenXC application can be implemented. Because there are many different ways to control access to the OpenXC service due to Android's inherent flexibility, we did not want to impose restrictions on this variability via our framework.

3.5 Android Services in OpenXC

There are two distinct services upon which OpenXC relies for proper communication with an Android application. The first of these is the VehicleService, which receives data via a physical interface, in our case USB, from the CAN translator. The VehicleService runs in a stand-alone Android process, separate from any Android application. Since only one Android application can connect to a USB device at any one time, this connection is handled by the VehicleService. The benefit of this design is that several OpenXC-enabled applications can run on an Android device simultaneously. The VehicleService is also responsible for processing the data and encapsulating it as Java objects before passing it on to a second service, the VehicleManager Service, which may be bound to multiple Android applications. It is possible for each OpenXC-enabled application to have its own connection to the VehicleManager. The OpenXC documentation recommends binding to this service with the BIND_AUTO_CREATE keyword, which keeps the service running as long as either it is explicitly started or there are one or more connections to it.

The VehicleManager is an abstraction of the VehicleService and its functions for the application developer. OpenXC has an enabler Android application, whose role is to start the VehicleService, which keeps the application developer from needing to manage it directly. The VehicleManager, on the other hand, is declared within the Android manifest and serves as an in-application service. It has two main modes of operation: synchronous and asynchronous. In synchronous mode, the application must make explicit requests for measurements from the service once the VehicleManager is bound and a connection to it has been established. An example of this request mode is given in Figure 6.

Figure 6

Synchronous Measurement Listener

```
try {
      VehicleSpeed measurement = (VehicleSpeed)
      vehicle.get(VehicleSpeed.class);
} catch(NoValueException e) {
      Log.w(TAG, "The vehicle may not have made the measurement
      yet");
} catch(UnrecognizedMeasurementTypeException e) {
      Log.w(TAG, "The measurement type was not recognized");
}
```

* This example is reproduced from
http://openxcplatform.com/android/api-guide.html

The asynchronous mode of operation allows an application to be kept up-to-date with any particular measurement. This involves registering a listener with the VehicleManager, an example of which is given in Figure 7.

Figure 7

Asynchronous Measurement Listener

```
VehicleSpeed.Listener listener = new VehicleSpeed.Listener() {
     public void receive(Measurement measurement) {
          final VehicleSpeed speed = (VehicleSpeed) measurement;
          // do stuff with the measurement
     }
};
vehicle.addListener(VehicleSpeed.class, listener);
```

* This example is reproduced from
http://openxcplatform.com/android/api-guide.html

4 Framework

4.1 Framework Goals

Our central goal for this framework is to provide for both human-to-machine and human-to-human interactions. As an exemplary human-to-human interaction, we implemented a messaging application, CarTalk. This application allows its end users to send and receive messages. We focused our implementation on the sending of text, but the system is readily extensible to send other forms of media as well. A good comparison to this module is the native Android messaging application. Our implementation, however, takes information from OpenXC to tailor message delivery to "safe" driving situations. CarTalk withholds message delivery if it senses that the driver is distracted with a critical task.

To account for human-to-machine interactions, we adapted CityBroswer, a point-of-interest query platform developed by the Spoken Language Systems group within CSAIL at MIT, for use with OpenXC and our particular framework. This latter constraint led to some interesting design choices when developing our framework.

4.2 Framework Design

For a framework of interconnected OpenXC applications, the first major defining network characteristic is centralization, or the lack thereof. A centralized approach was considered, whereby all accesses to the OpenXC API would occur through a single access point, in the form of a central, or head, node. Given that OpenXC uses an enabler application in its base implementation, this seemed like a logical enhancement.

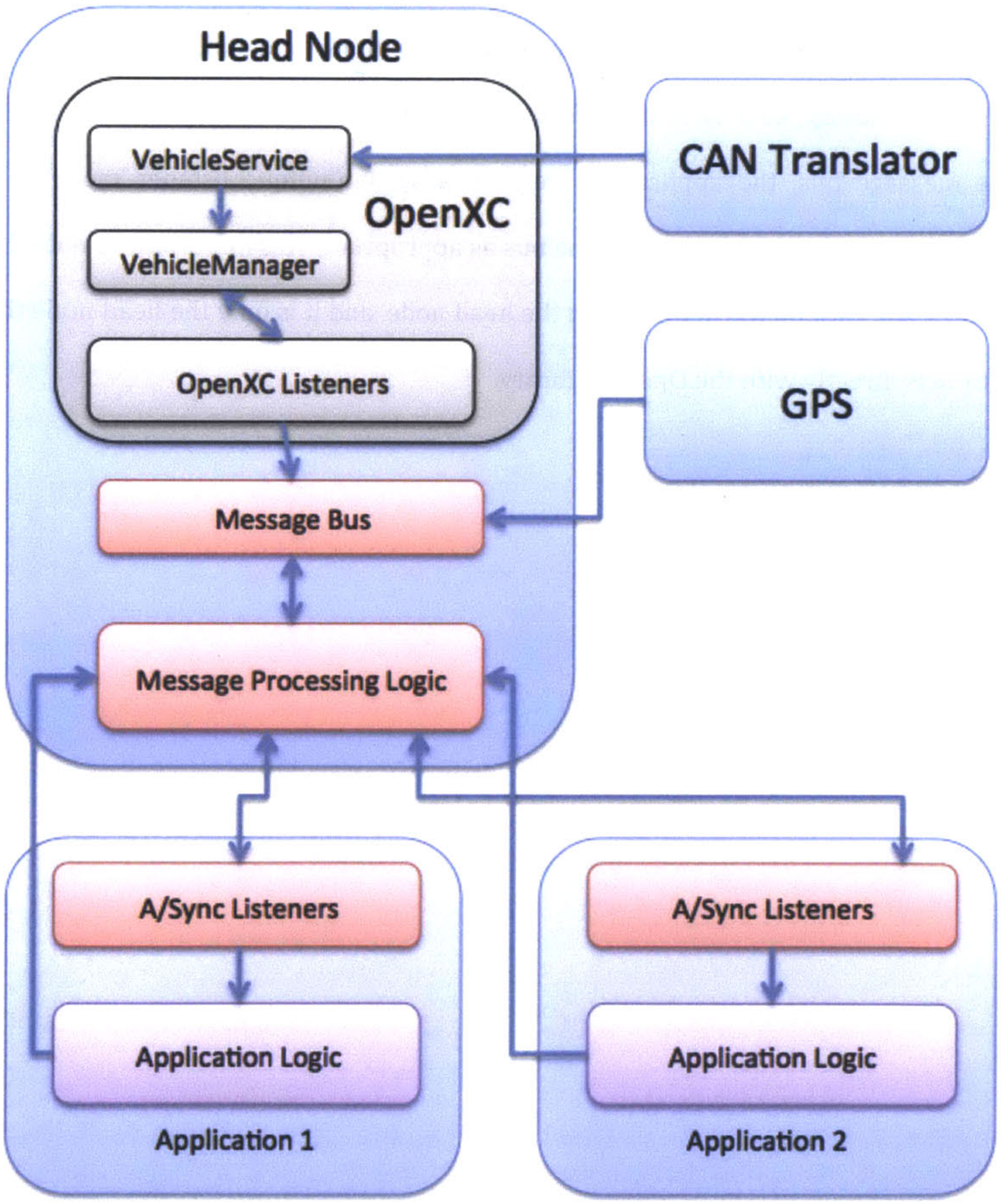
4.3 Centralized Design

A centralized framework design binds both the VehicleService and the VehicleManager to a single point-of-access. This application-as-a-service effectively controls all of the OpenXC-related accesses used by the other applications in the system. Additionally, the head node serves as a processing station for all inter-node communication.

For a centralized approach, an Android application (which we will designate a node) has two main API calls that it can make with respect to the transmission of data. A node can elect to transmit information, in which an Android Intent is sent from the transmitting node to the head node. The data item itself can be attached to the Intent via the putExtra() method call. With this transmission scenario, the head node acts as a data repository, keeping a mapping of applications to data elements. However, the head node may store a reference to the transmitted data, rather than the data itself, to reduce overhead. This can be accomplished with Android's ContentProviders. ContentProviders provide references to resources via unique identifiers in the form of content URIs. While this implementation is beyond the scope of our discussion, it is important to note that there are many possible variations for data transmission within a centralized network structure. Requests for reception of data in this network must, as with transmission, flow through the head node. This is possible with a subscription model, whereby the head node accepts the attachment of listeners from other individual nodes. The listener model preserves both the synchronous and asynchronous modes of operation of the OpenXC framework. In addition, there is no distinction between requests for OpenXC data and for data from other applications. Both of these types of requests are encapsulated by our proposed centralized design.

Figure 8

Centralized Framework Design



The preceding figure shows the layout of the centralized design for our application framework. The operation of OpenXC is controlled via a head node, with the head node taking on the responsibilities regularly reserved for the OpenXC enabler application. The major characteristic of this design is that individual applications subscribe to a message bus to receive content from any outside sources, including OpenXC readings, GPS data, and information from other applications. To connect to the message bus, the applications rely on a system-unique listener model, and subscribe to or request data from the bus as appropriate. It is important to note that all messages are forwarded through the head node, and it is only the head node that interacts directly with the OpenXC library.

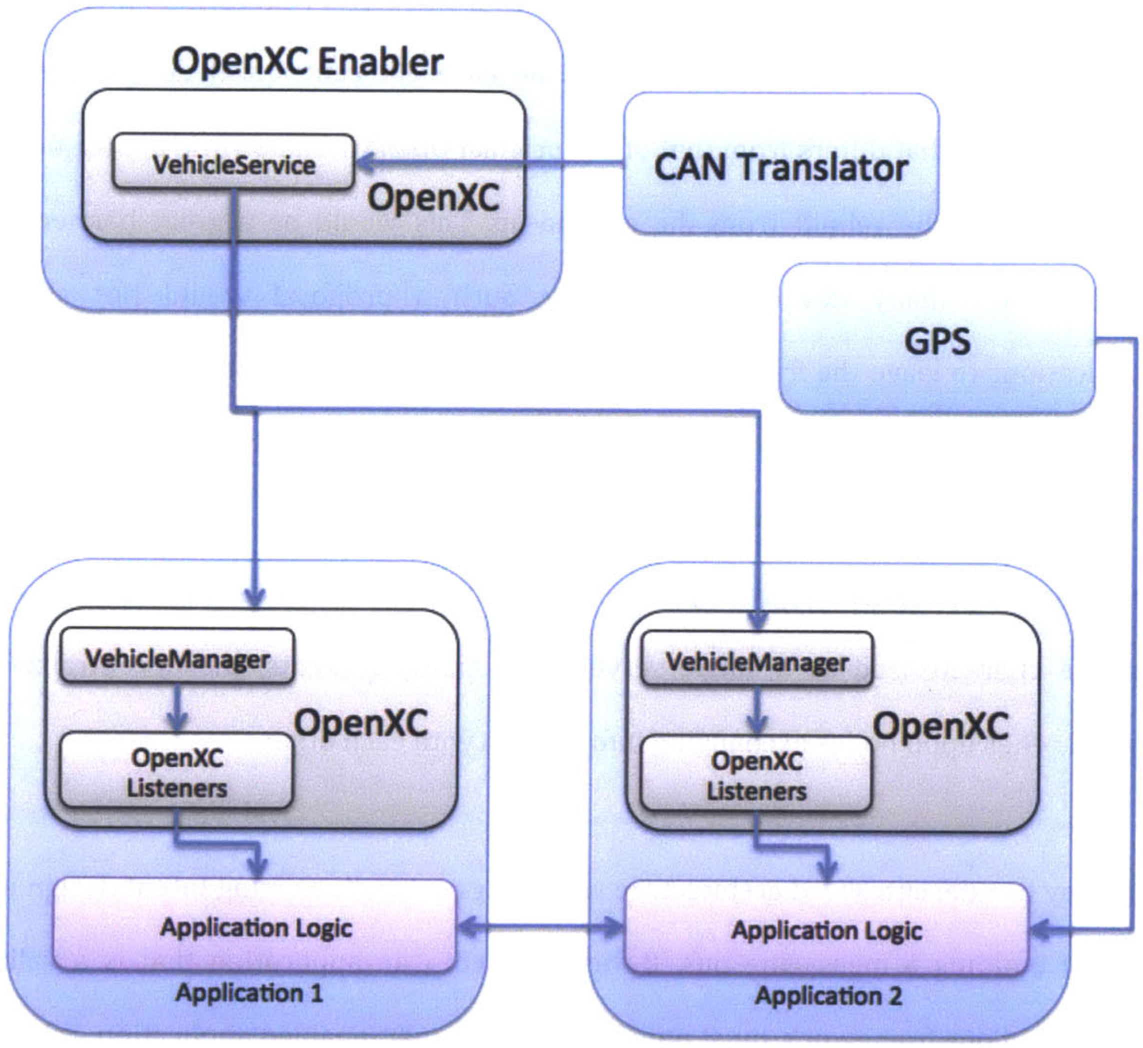
4.4 Decentralized Design

There are, however, some strong drawbacks to a centralized proposal. Our main concern with a centralized framework was the difficulty in incorporating existing stand-alone OpenXC-enabled applications. Because our centralized design uses a listener model that differs from that of the original OpenXC framework, applications would need to be rebuilt from the ground-up. This would be a great barrier to adoption for many developers. Similarly, such a proposal would not allow applications to leave the framework design without significant restructuring on the part of the developer. We wanted applications to be useable as independent modules that could interconnect and share information, but could also act as standalone applications with only minor modification. Where the centralized model can be characterized as a single server with multiple clients, the decentralized model relies upon nodes exchanging information with each other.

The key to a decentralized architecture was using Android's existing Intent design in lieu of building a messaging bus. By using Intents, an application that is a valid member of our framework, must only be a valid OpenXC-enabled application. With a decentralized framework, each application uses the OpenXC library independently. We rely upon the VehicleService to sort out contention issues relating to the access of OpenXC messages. Each application is allowed to use the VehicleManager service independently. Applications are responsible for providing an API for their data.

Figure 9

Decentralized Framework Design



This figure shows the layout for the decentralized design of our application suite. This design is much less restrictive to applications than is the centralized approach. Each application interacts with OpenXC in the traditional way, directly via the

VehicleManager and listeners. We have included the OpenXC enabler application in this figure, as it is an integral plart of the OpenXC libraray. However, it is possible for individual applications to internalize this logic. Instead of using a messaging bus at a central node, we rely on the underlying Android OS to transfer data, using Intents. Applications also connect directly to external data sources, such as a GPS or gyroscope. In this sense, applications have the ability to behave in a stand-alone fashion, or can be combined with others to form a suite.

5 Applications

5.1 CarTalk

5.1.1 The CarTalk Application

To showcase the benefits of OpenXC-enabled applications, we have developed a messaging application for the Android platform. The main distinction between our application and a standard messaging app is the use of vehicular feedback to determine driver awareness. By monitoring the data provided by the OpenXC library, we can make qualified assumptions as to what a driver is doing. This allows us to distinguish high-concentration versus low-concentration events, and provide notifications in only the latter states. In addition, the CarTalk application, when coupled with our point-of-interest browser, allows the driver to perform lookup and share tasks while verbally interacting with their Android device.

The CarTalk application consists of two core parts. There is the client-side Android application, which provides the UI for the user and is also responsible for using the OpenXC library. The client-side portion performs all processing with respect to monitoring driver awareness. All of the logic pertaining to analysis of the OpenXC messages is performed on the client side. The second part of CarTalk is the server, which acts as a message repository. Its role is to organize and to store messages that are transmitted from multiple Android client devices.

5.1.2 CarTalkServer

5.1.2.1 Overview

The CarTalkServer is a Google App Engine web application implemented in Java. It consists of a set of Java Servlets, each of which responds to a specific HTTP GET or POST request. We developed several versions of the server component, with differing levels of security for user data. The goal, however, was to emphasize functionality in the proof-of-concept form. We are currently in the process of upgrading the server to have a security level robust enough to be deployed in a production environment. One of the upgrades involves support for server-client sessions, where a user must log in to the server to establish a connection.

The CarTalkServer makes use of Google's App Engine Datastore, a NoSQL schemaless object datastore. The App Engine Datastore provides a query engine, and transactions appear as atomic operations. Additionally, the App Engine Datastore handles cross-machine data replication. It guarantees strong consistency for all read queries, meaning that reads from parallel reads are ordered sequentially, so that a consistent state is observed. The datastore observes an eventual consistency model for all other queries. To support application scalability the App Engine Datastore relies upon replication between different data centers.

5.1.2.2 Entities

The datastore contains data objects called entities, which each have one or more properties. A property can be thought of as a key-value pair. The value for the property may be one of many data types, including a reference to another entity. Each entity is of a particular kind, which is a categorization used for queries. Each entity also possesses a unique key. The key is permanent, and may be assigned either at the application-logic level, or by the datastore itself.

5.1.2.3 Handling Login

The CarTalkServer currently provides three separate functionalities. Interactions with the server are user-based, meaning that a client must establish a connection as a specific user before performing any other transactions. This logic is implemented within the LoginServlet class. The LoginServlet responds to POST requests to the url `http://cartalkserver.appspot.com/login`. The client must supply a username parameter. If the username parameter exists and a valid password hash is also supplied, then the server returns an alphanumeric identifier, which can be used as a session key. This was implemented with the ability to maintain session-based interactions between client devices and the server. Upon receipt of a POST request, the LoginServlet queries the datastore for an entity with a matching username. If one is found, a session identifier is returned. Otherwise, the servlet creates the necessary entity in the datastore before returning a session identifier. Our model relies upon the invariant that all usernames are unique, as they are used as datastore keys for a given user.

5.1.2.4 Sending Messages from a Client

The second functionality that the server provides is the ability for clients to exchange data with one another. The server acts as a go-between in these exchanges. Information is uploaded to the server by the sending client, and the receiving client is responsible for polling the server for updates. The servlet responsible for implementing this functionality is the SendShareableServlet. While the SendShareableServlet was designed abstractly, with the ability to send varying types of data, we will focus on our proof-of-concept implementation, which transfers text messages. The SendShareableServlet responds to POST requests to the url `http://cartalkserver.appspot.com/send`. The sending client must include values for the “from”, “to”, “data”, and “date” properties that will be associated with the message. The servlet’s first action is to validate the existence of both the sending client’s and the receiving client’s entities in the datastore. This ensures that messages will be received by a valid client. The servlet executes this check by performing two read queries upon the datastore, one for each client. If either client’s entity is null, the message transaction is voided, and an appropriate failure message is returned. Otherwise, the servlet creates a new entity of the "Shareable" type to represent the message, and fills in its properties accordingly.

5.1.2.5 Receiving Messages

The third functionality provided by the server is the ability to view the messages sent to a particular user. The logic for this is housed within the GetShareablesServlet. This servlet responds to GET requests sent to the url `http://cartalkserver.appspot.com/get`. The client must include the desired username as a parameter for the request. Upon receipt of a new request, the servlet checks the requester's username with that of the current session. Each query must have an associated session, which is created during the login sequence. If the two usernames match, then the GetShareablesServlet queries the datastore for all messages sent to the requested username. Currently, as we only deal with text messages, the servlet response with a text representation of the entities representing the messages sent to the username. The entities' representations include all of the properties of the messages, with any empty key-value pairs represented by an empty String.

5.1.3 CarTalk

5.1.3.1 Overview

The second half of the CarTalk application is the Android client. This portion of the application is responsible for the user experience. It is also on the client device that we perform all processing related to the OpenXC library. The CarTalk application consists of a set of Android Activities, each of which is responsible for a different user action. As our current version of the CarTalk application supports text messaging, we have four distinct user actions. These actions are as follows: logging into the service, viewing a "home screen" of all received messages, viewing an individual message, and sending a message.

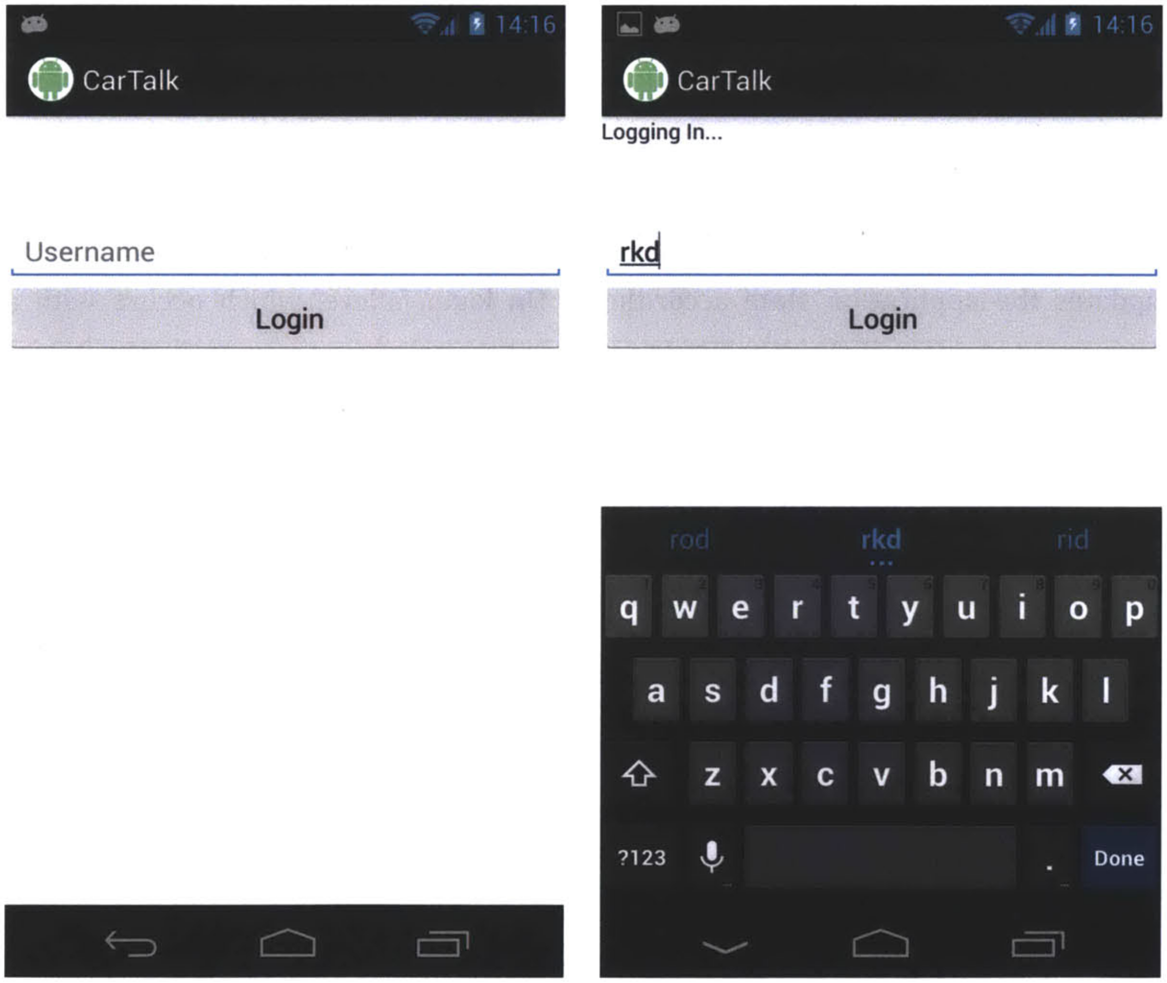
5.1.3.2 Logging In and Asynchronous Tasks

The login Activity, named CarTalkMain, serves as a gateway for the rest of the CarTalk application. Rather than assigning a one-to-one relationship between users and devices, we allow multiple users to use the same device. The goal of allowing this functionality to be user-visible is to allow separate accounts for when users are outside a vehicle, but still allowing passengers access to a terminal that has been permanently installed within a vehicle by a different user. CarTalkMain provides asynchronous login logic, which keeps the Activity from blocking Android's main (UI) thread. This is the required logic design when performing a request-response interaction between an Android Activity and an outside server that needs to eventually update the UI. We repeat this design pattern throughout the CarTalk application, whenever a response from the CarTalkServer is necessary. We make use of Android's AsyncTask, an abstract class that allows us to perform work in a background thread. In the case of CarTalkMain, this work is network related, sending off a user's textual input for login approval by the CarTalkServer. We needed the AsyncTask in this situation to inform the user that work was, in fact, being done and that the application had not failed silently. This behavior serves to provide a better user experience. Upon receipt of the server's response, the client updates the application state accordingly. On login failure, which occurs with a mismatch of credentials, the UI is updated with a "Login Failed" alert. Otherwise, the user is allowed to proceed to the next Android Activity, which provides a different set of functionality. Because our application relies on a connection to the CarTalkServer, we require that the mobile device have a network connection. If a network connection drops, CarTalk can no longer receive responses from its asynchronous requests, resulting in an inability to send or receive new messages. Login is considered successful if either there is a username/password match with an existing account, or there is no username match with an existing account. In the latter case, a new account is created with the given parameters. We allow cases where it is possible for users to bypass the login screen completely, even if the

CarTalk application is not currently running. We make use of this functionality in a pass-through from the point-of-interest CityBrowser to CarTalk, which works as long as the user has a valid CarTalk username associated with their CityBrowser application.

Figure 10

The Login Activity



The Login Activity, named CarTalkMain in our implementation.

5.1.3.3 The Home Activity

After a successful login, the user is presented with the UI controlled by the Home Activity. It is this Activity that makes use of the OpenXC library. The role of the Home Activity is to give the user a summary of their received messages. However, this must be done only when the user is not preoccupied with a critical driving task. The Home Activity follows the same AsyncTask design pattern that we outlined for CarTalkMain and our login process. We use this pattern to received new messages from the CarTalkServer. The messages are stored in a buffer object, but they are not immediately displayed on the UI. This latter step is performed after checking for driver awareness.

5.1.3.4 Gauging Driver Awareness

Checking for driver awareness is the key feature that distinguishes CarTalk from a traditional messaging application. To accomplish this, the Home Activity makes use of the OpenXC library with asynchronous listeners. A set of OpenXC-related variables is maintained in a dynamic state; these variables are updated by the asynchronous listeners whenever an updated value is available via the VehicleManager. The variables are made available to a TimerTask, whose purpose is to recurrently predict driver state. Currently, the output of this task is binary, with the outcome being either the delivery or withholding of newly received messages from the server. For future implementations, we hope to extend this functionality to distinguish between different types of messages and notifications. The goal is to match different notifications with different awareness states, as monitored by OpenXC. For example, a voice message does not necessarily need the same level of awareness as does a text message. We hope to perform a set of user studies to better quantify awareness states, and the notifications that are applicable in each of them.

Our current implementation provides a rudimentary classification mechanism. We monitor vehicle speed and brake pedal status to determine whether a driver is stopped at a light. If the vehicle is stationary and the brake is being held, we allow CarTalk to present the driver with new messages.

Figure 11

Rudimentary Awareness Classifier

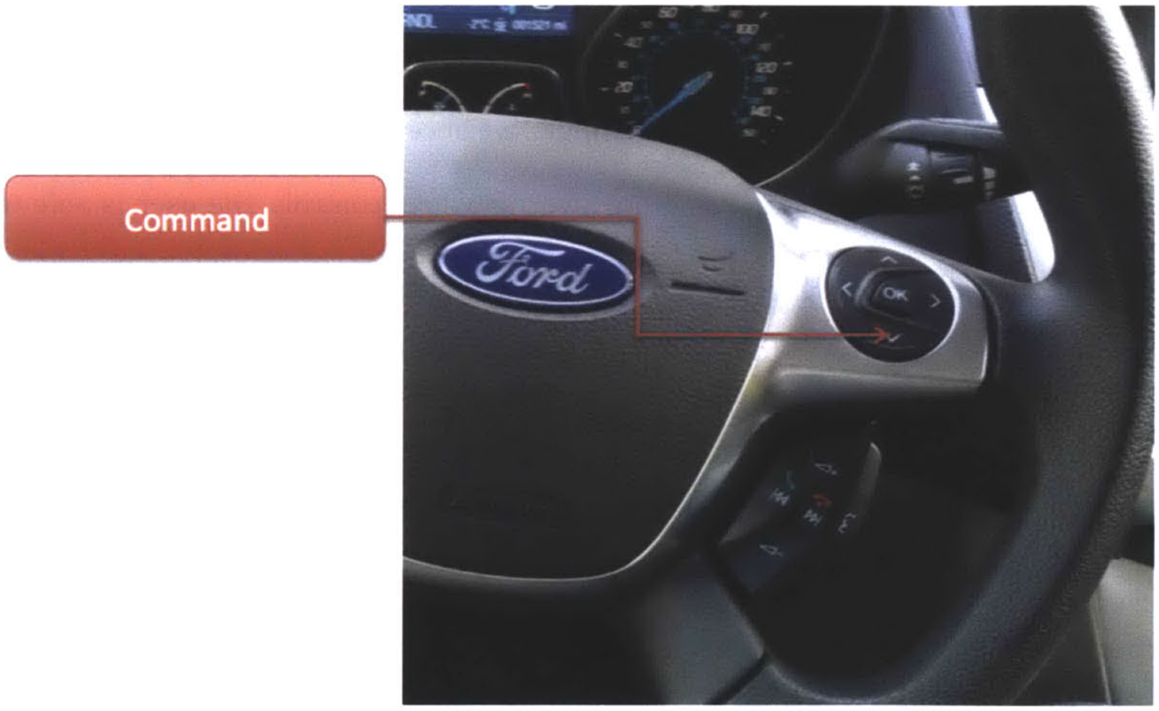
```
if (mVehicleSpeed < 10 && mVehicleBrakeStatus) {
	// Get our messages
	GetMessages getMessages = new GetMessages();
	getMessages.execute(user);
}
```

5.1.3.5 Viewing and Sending Messages

From the Home Activity, the user has the option to view an individual message in detail, or to send their own message. The ViewMessage Activity is unremarkable with respect to the features of the application, other than to focus the user's attention to a single message with the limited UI real estate available on mobile devices. The SendMessage Activity, on the other hand, serves as a model of human-machine interaction. Rather than using the physical mobile device to input message details, a driver can use controls on the steering wheel and voice for input. To monitor the steering wheel for control inputs, we once again use the OpenXC library, with asynchronous listeners. Using the steering wheel controls, the user is able to activate speech recognition, as well as send their message. We rely on Android's built-in speech recognizer to handle this task. A user can use the speech feature to either issue commands to the application, or to dictate portions of their message. CarTalk currently supports two distinct commands, which must be prefaced with the phrase "CarTalk" to distinguish them from message or username dictation. The first of these commands is a "send" command, whose function is akin to pushing the "send" button on the application's Android UI. The second command is a "paste" command, and is used to transfer the contents of the clipboard into the message body. The reason that the command is prefaced with the phrase "CarTalk" is so that it can be used inline with normal dictation. Like the Home and CarTalkMain Activities, the SendMessage Activity makes use of the AsyncTask design to allow UI feedback for message status. We notify the user when a message is sent successfully, or if transmission failed.

Figure 12

CarTalk Steering Controls Mapping

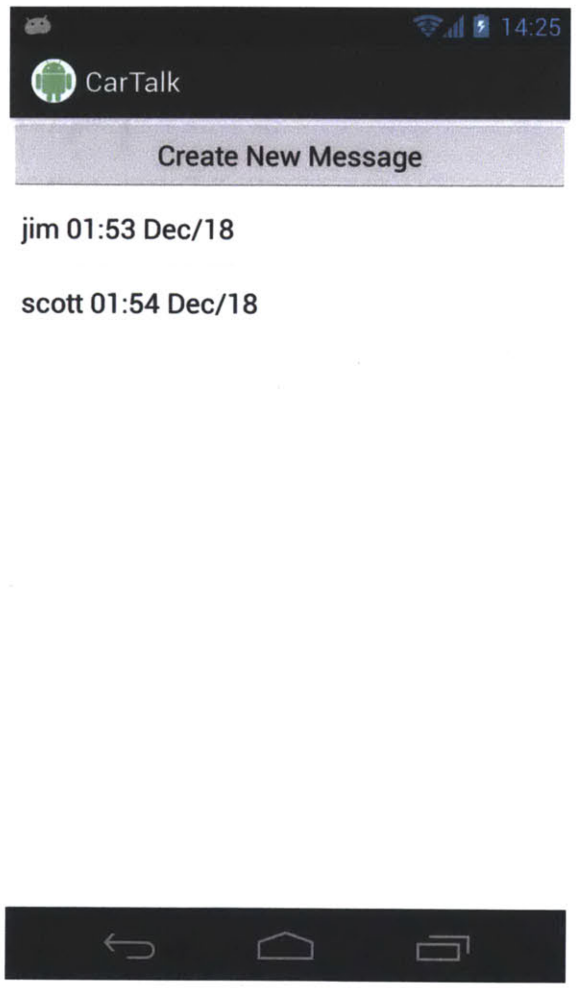


For the CarTalk application, we focused on simplicity when mapping application functions to steering wheel buttons. Rather than use several buttons and map an action to each one, we only employed a single steering wheel button. When pressed, this button activates the voice recognition engine on the Android device. This allows the user to dictate a larger set of commands. Please see section A1 for example usage.

Figure 13

The Home Activity

14:17
CarTalk
Create New Message



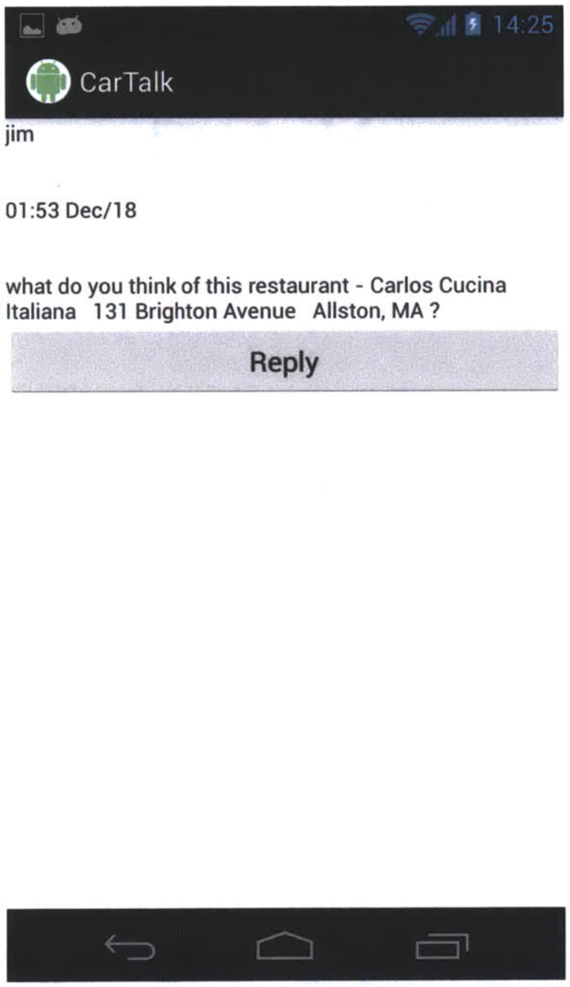
jim 01:53 Dec/18

scott 01:54 Dec/18

The Home Activity, before and after message updates.

Figure 14

The ViewMessage Activity



jim

01:53 Dec/18

what do you think of this restaurant - Carlos Cucina Italiana 131 Brighton Avenue Allston, MA ?

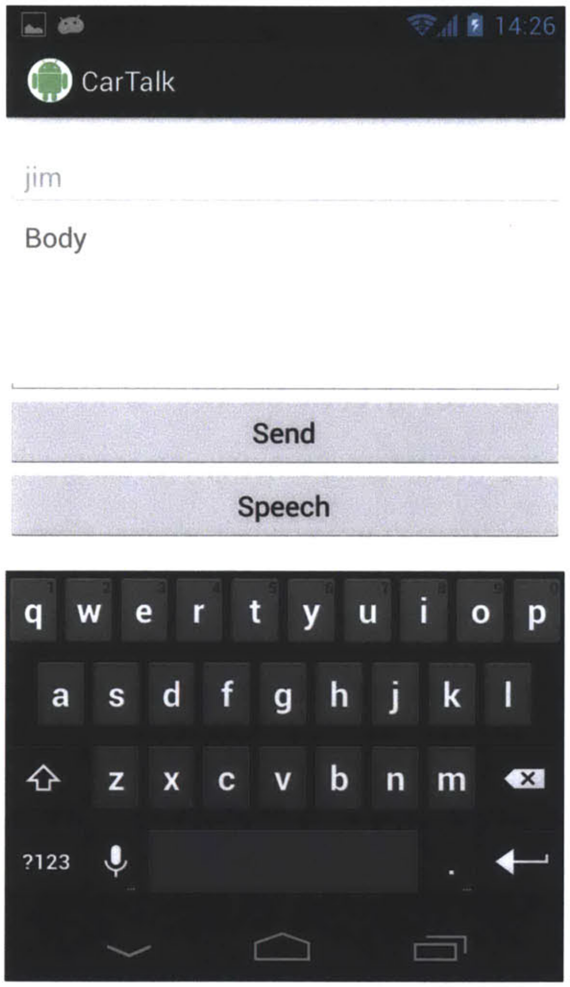
Viewing an individual message

5.1.3.6 CarTalk Summary

The two key components of the CarTalk mobile application are the Home Activity and the SendMessage Activity, which we use as two proof-of-concept implementations. The Home Activity demonstrates the use of OpenXC to check for driver awareness when presenting notifications. Its goal is to provide a safer way to enable human-to-human interactions while in a vehicle. We wish to demonstrate that time spent commuting can also be time spent engaging with others, within reason. The SendMessage Activity takes the same idea of safe communication, but applies it differently. The SendMessage Activity is designed to make the physical device's screen redundant, as all of its operations can be achieved with a combination of steering wheel controls and voice inputs. This allows a driver to focus their visual attention on the road ahead, instead of staring at a screen.

Figure 15

The SendMessage Activity



Replying to a message

5.2 CityBrowser

5.2.1 The CityBrowser Application

The CityBrowser application started out as an existing web application of the same name, developed at the Computer Science and Artificial Intelligence Laboratory at MIT. The functionality that it provides is a human-to-machine interaction, as opposed to the human-to-human interaction of CarTalk. CityBrowser takes voice queries as user input, and responds with a set of points of interest. While the browser can be configured to query several datasets, including flights and movies, we worked with restaurant queries. Because the original application is web-based, there were numerous design choices that came with adapting CityBrowser for use with OpenXC and our framework. The original web component of CityBrowser incorporates voice recognition from the WAMI project. WAMI is an open source Javascript API that allows projects to integrate speech recognition. Because of this, our Android application needed to be able to support both the WAMI speech recognition engine as well as the built in Android recognition engine. Most importantly, we needed a method to allow OpenXC, a Java library, to interface with the web technologies of the point-of-interest application.

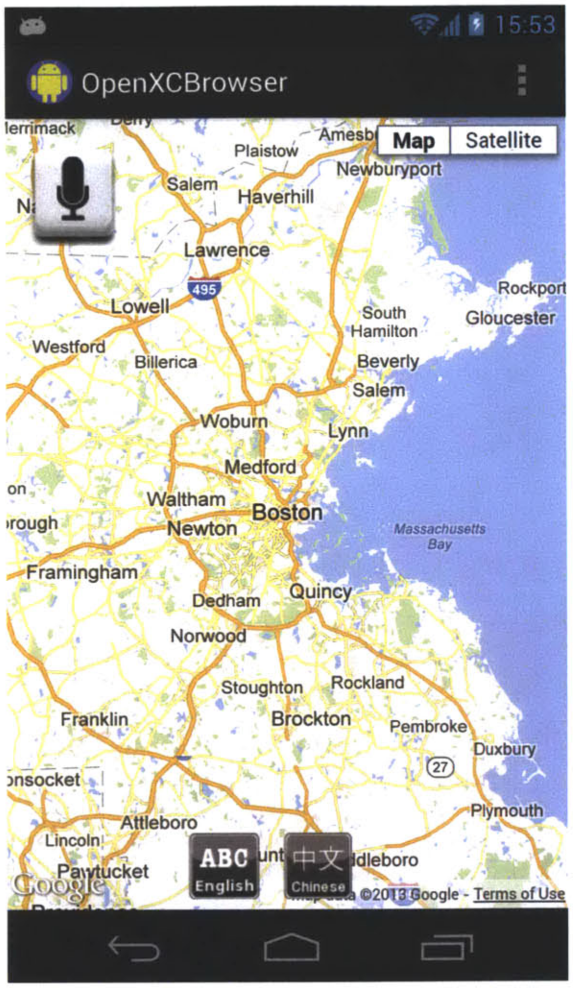
5.2.2 The Asgard City Browser

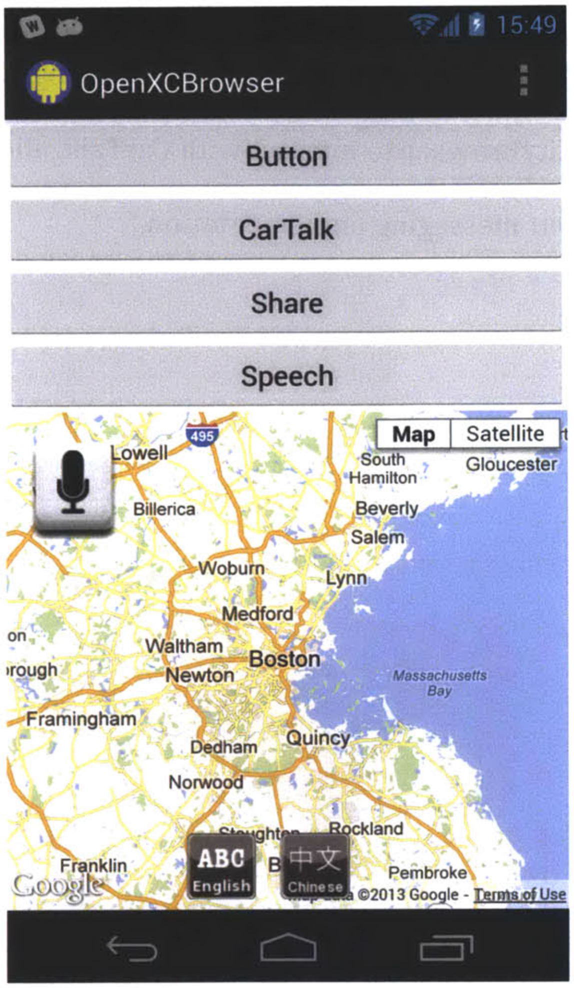
The core of CityBrowser is a point-of-interest query web application. It employs standard web technologies, notably HTML and Javascript. To allow for the display of web-based content on Android devices, Android applications provide the WebView, a class that delivers the necessary functionality. The WebView is an Android View object that uses the WebKit rendering engine to display web pages and includes methods to manipulate these pages. These methods include navigation, zoom, and text search, all akin to methods found in a standard Android mobile browser. A basic WebView provides no browser-like widgets and does not enable Javascript on the presented pages. However, there are many features of WebViews that allowed us to combine tranditional web technologies with our OpenXC framework, without requiring any modifications on the server-side for CityBrowser.

The core of CityBrowser, known as the Asgard City Browser, is currently located at `http://web.sls.csail.mit.edu/CityBrowser`. The mobile-optimized version of the Asgard City Browser is located at `http://web.sls.csail.mit.edu/CityBrowser?mobile=true`. The mobile version of the site has a slightly different user interface for mobile devices. To start a query, a user must press a button on the web application. This starts the audio capture for the user's speech. To end the capture, the user must press the button a second time. At that point, the audio is sent off to the server, where it is processed. The server responds with a set of restaurants that match the parameters presented in the user's request. These are displayed on a map, and the results are also read aloud to the user. We added several key components for the UI that increased the capabilities of CityBrowser. We modified the application to allow it to be given voice commands, not just queries. Additionally, we added support for OpenXC, which allows the application to take steering wheel inputs. Lastly, we added support for CityBrowser to interact with CarTalk, allowing users to share their restaurants over our messaging implementation.

Figure 16

The CityBrowser UI

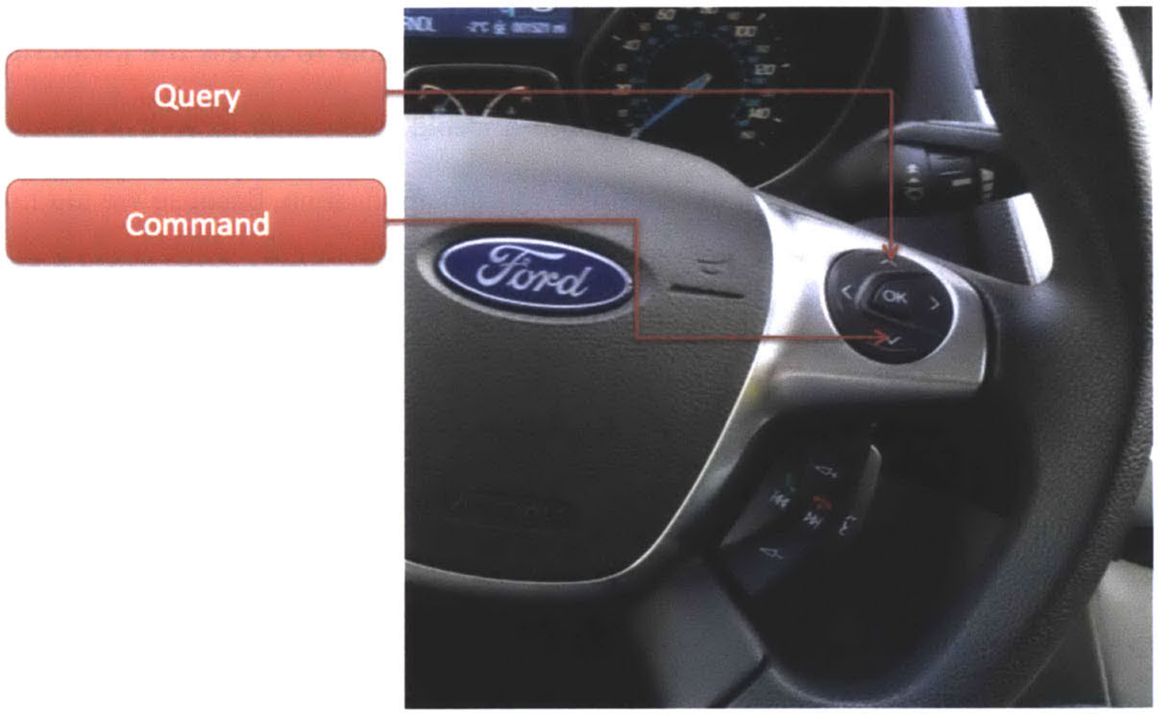




This figure presents two version of the CityBrowser UI. The figure on the right shows the debug UI, featuring buttons that are normally handled via speech or steering wheel commands. Later figures will feature the debug UI to show feature availability.

Figure 17

CityBrowser Steering Controls Mapping

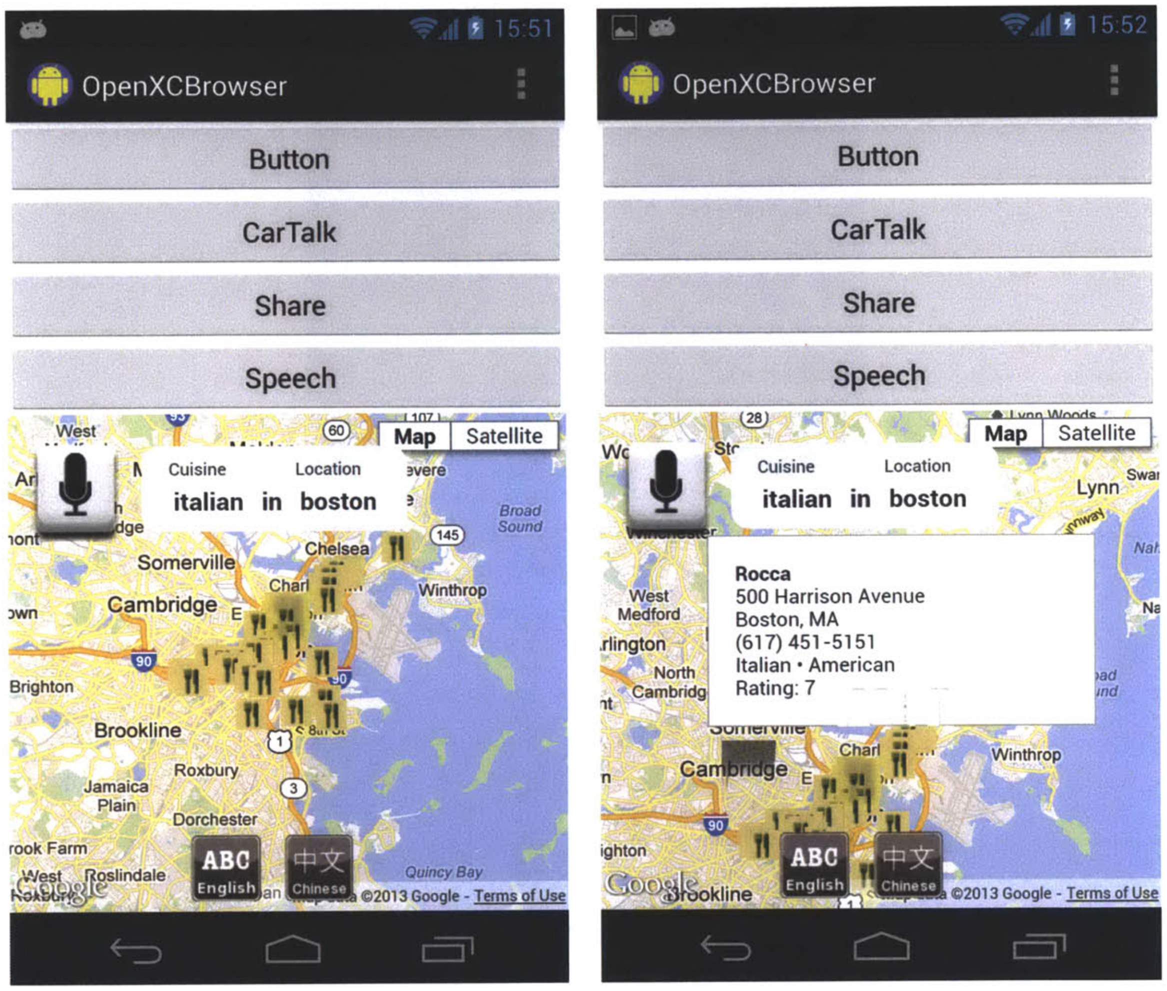


This figure shows the mapping of commands for the CityBrowser application. As with the CarTalk application, we focused on simplicity for our mapping. Thus, we only rely upon two of the steering wheel buttons. One, labeled above as "Query," is used to initiate and to terminate voice queries to the underlying Asgard City Browser, providing point-of-interest functionality. The other button, labeled "Command" above, is used to activate the voice recognition engine for CityBrowser. This allows the user to dictate commands to the application, as we had allowed for CarTalk. Please see section A1 for example usage.

Because the Asgard City Browser is a web-based tool, we wanted to maintain separation between our OpenXC-enabled implementation and the original application. The goal was to have no server-side modifications, so that a request from our modified application would be indistinguishable from that of the original implementation. For this to happen, we needed to be able to access the Javascript methods within the web application from our Android application, as well as to have the web application call methods from our Android application. Because we did not want to modify the web application on the server-side, we decided to establish a "compatibility layer" between the web application and the rest of our OpenXC framework. This decision kept us from using a centralized design for our framework, because doing so would severely complicate the logic contained within CityBrowser. The compatibility layer contains all of the logic necessary for CityBrowser to use OpenXC, as well as to pass information to other applications in our framework.

Figure 18

Querying With CityBrowser

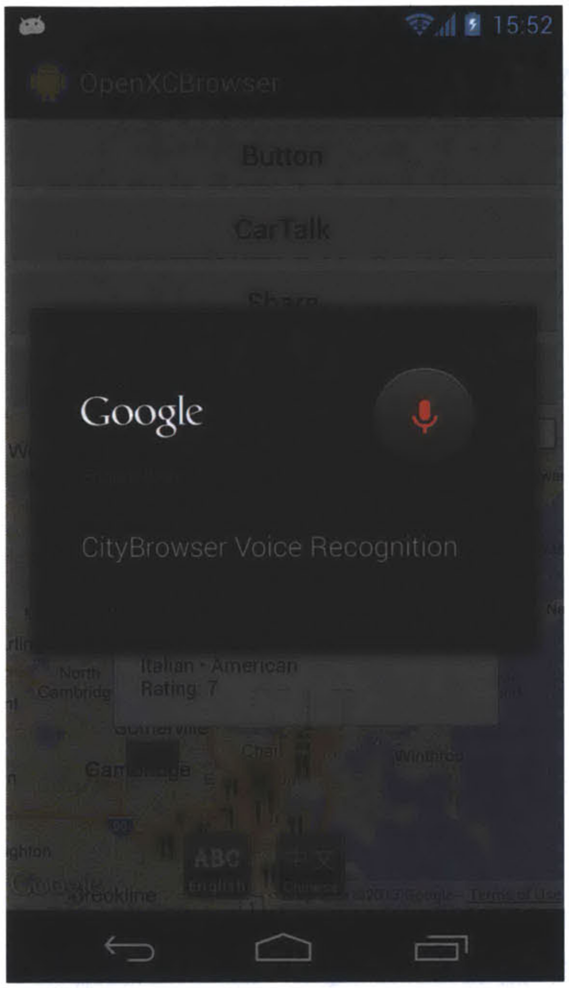


Rocca
500 Harrison Avenue
Boston, MA
(617) 451-5151
Italian • American
Rating: 7

These two figures demonstrate the act of querying the CityBrowser application for a point of interest. This process is identical to that of the original Asgard City Browser, except that it can be initiated via steering wheel controls.

Figure 19

Using the Google Voice Recognizer in CityBrowser



This figure shows the Google Voice Recognizer, launched via steering-wheel controls. This allows the user to input a variety of speech commands.

5.2.3 HijackJSInterface

To be able to pass information from the web application to Android, we use a Javascript Interface, a construct that can be added to a WebView. A Javascript Interface allows us to inject Java objects into a webpage's Javascript context, so that they can be accessed by Javascript in the page. To attempt to future-proof our Web-to-Android bridge, we have assigned our interface to the "hijack" namespace, and have made it easy to add additional methods to the existing interface. At a minimum, we needed a way to get the point-of-interest data from the WebView and into a Java representation. Similarly, we needed a way to update this data for different queries. For CityBrowser, we currently make use of a just a single method. This sendToAndroid() method takes a String as an argument and, after some filtering, assigns the String text to a Java object. While a typical query may return several restaurant options, these options are overlaid on a map as markers, as which point the user has the option of selecting the best option. It is after this last step that we grab the data in question, which keeps us from having to duplicate logic that already occurs on the server side. For instance, our approach means that we do not need to keep track of which restaurant marker a user clicks on; neither do we need to keep track of all of the returned results. However, it is worth noting that our Javascript Interface implementation is easily extensible, so it is trivial to add in new methods to grab any possible data from the WebView.

Figure 20

The HijackJSInterface

```
public class HijackJSInterface {
      public HijackJSInterface() {
      }

      public void sendToAndroid(String text) {
            text = text.replaceAll("<br>", " \n ");
            text = text.replaceAll("</(\\w)+>", " \n ");
            text = text.replaceAll("<(\\w)+>", "");
            text = text.replaceAll("&bull;", ",");
            currentPOI = text;
            share.setClickable(true);
            share.setActivated(true);
            Log.v(TAG,text);
      }
}


@Override
public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      mWebView.addJavascriptInterface(new HijackJSInterface(),

      "hijack");
}
```

This figure gives an example of how the HijackJSInterface class is used. During the Activity's onCreate() method, the HijackJSInterface is added as a Javascript Interface to the WebView that contains the Asgard City Browser. The interface is assigned to the "hijack" scope, which gives a consistent reference point. For this example, the sendToAndroid() method is available from the Asgard City Browser Javascript via hijack.sendToAndroid(). This method effectively turns a Javascript object into a Java object.

Having solved the issue of providing Java methods for the WebView to call via our Javascript Interface, we still had the problem of calling those methods. The problem resided with our goal of allowing no server-side logic changes, even though the Asgard City Browser server is responsible for delivering the client-side Javascript logic. To get around this issue, we resorted to Javascript injection. We injected Javascript code into the web-application via the WebView's loadUrl() method. It is important to note that our method relied upon changing logic on the client side, leaving the logic supplied by the Asgard City Browser server unchanged. Our injections came in two particular flavors.

5.2.4 Patching JS Functions

Our first case deals with the method of initiating audio capture when the user is not employing the physical mobile device. To accomplish this, we have created a Java method that injects Javascript into the WebView that simulates a click on the HTML element that represents the initiation button. In this case, we use Javascript injection to call existing methods in the WebView, without the need for processing any results. We refer to this type of code injection as "patching," since no existing methods are overridden by this function.

Figure 21

An Injected Javacsript Patch

```
loadUrl("javascript: $('#wami-microphone > div > div >
div').click();");
```

The above shows a code example for an injected Javascript "patch". The logic presented above is used for simulating a click in the WebView from our Android Java code. We inject Javascript code via the WebView's loadUrl() method, without the need for processing a return value.

5.2.5 Hijacking JS Functions

Our second case is different from the first in that it deals with us getting data back from the WebView. While we have previously described our construction of the Javascript Interface, there did not exist a method to call these new functions from the original logic. To resolve this issue, we chose to hijack an existing method within the Javascript logic. We refer to this type of injection as a "hijack" because we override an existing method within the client-side Asgard City Browser logic. To accomplish a hijack, we make use of the fact that Javascript allows for the overloading of functions, and that a given function can be reassigned to a new variable. For this technique to succeed, hijacked functions must reside at the same scope as the original functions that they replace. Furthermore, the hijacking function must be of the same name as the original function. Because our goal was to return point-of-interest data from the WebView into a Java object, we targeted an existing function that had this data available. The function in question is named addMarker(), and serves to place a marker onto the map to represent the location of a restaurant. In addition to placing a marker on the map, the addMarker() function attached an on-click listener to the marker. Knowing that a user has set off a given listener allows us to determine exactly which restaurant they are currently interested in, and this is the data that we wish to pass on to our Java representation. To do this, we injected Javascript code into the WebView that overwrote the original addMarker() function, replacing it with a method that preserved the original functionality while simultaneously calling our new method from the "hijack" scope. In this way, the functionality of the original logic is not impaired, but our new method is inserted appropriately.

Figure 22

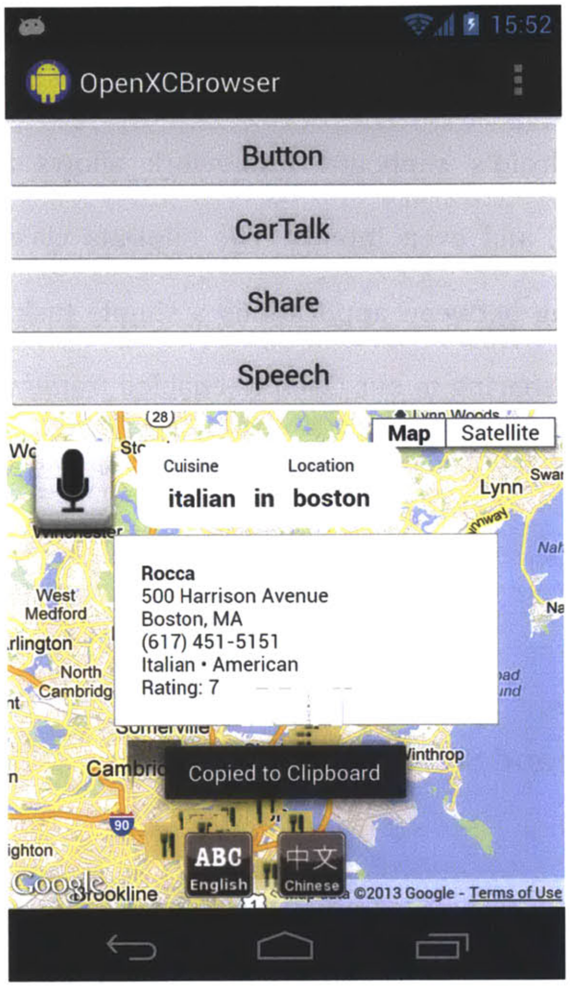
A Hijacked Javascript Function

```
private void hijackAddMarker() {
     String sled = "javascript: var originalAddMarker = " +
          "     addMarker; function addMarker(location, " +
          "     contentString, imgUrl, title, _oid) {" +
          "     var marker = new google.maps.Marker({"+
          "     position : location," +
          "     map : map," +
          "     icon : imgUrl," +
          "     title : title," +
          "     optimized : false" +
          "});" +
          "marker._oid = _oid;" +
          " " +
          "google.maps.event.addListener(marker, 'click', " +
          "     function () {" +
          "     infowindow.setContent(contentString);" +
          "     infowindow.open(map, marker);" +
          "     fireCityEvent('MARKER_CLICKED', marker);" +
          "     hijack.sendToAndroid(contentString);" +
          "});" +
          " " +
          "markersArray.push(marker);" +
          "}";
     loadUrl(sled);
}
```

The above example illustrates an injected Javascript "hijack." This particular example demonstrates the overloading of the addMarker() function. The key change here is the inclusion of the hijack.sendToAndroid() call within the body of the maps event listener. The role of the original addMarker() function is maintained with this modification, but new logic is now introduced into the listener's on-click sequence. This type of injection allows us to pass objects from Javascript into the Java of our Android application.

Figure 23

Sharing in CityBrowser



Rocca
500 Harrison Avenue
Boston, MA
(617) 451-5151
Italian • American
Rating: 7

This figure demonstrates the use of a hijacked function, along with the HijackInterface, to transfer point-of-interest data from within the WebView to the Android clipboard.

5.2.6 Sharing in CityBrowser

Once we have captured a user's point-of-interest, we allow them to exchange this information with others via CarTalk. To accomplish this, we employ existing Android tools. To share data, we rely upon Android's clipboard and the act of sending Intents. Android's clipboard framework allows applications to share numerous data types, and even Intents. The ClipData class and its convenience methods make sharing between applications a simple task, and we adapted this existing structure for sharing in our OpenXC-enabled framework. For our proof-of-concept implementation, however, we rely upon the clipboard solely to share text.

5.2.7 CityBrowser and Steering Controls

To show the benefits given by OpenXC-enabling this human-to-machine application, we incorporated steering controls and voice commands into CityBrowser. This was done in the same fashion as for CarTalk. The user is able to start and end CityBrowser's audio capture via the push of a steering-wheel-mounted button. To make distinguishing between queries and commands easier, there are two separate buttons that we make use of on the steering wheel. One serves to initiate queries, while the other is a directive to the program to listen to a command. CityBrowser responds to verbal "copy" requests, and can also recognize commands to launch

CarTalk. We structured our program this way to allow seamless transitions between CityBrowser and CarTalk, such that the user can focus on the task at hand, and not on which application they need.

6 Conclusion

6.1 Summary

With this work, we have developed a proof-of-concept application suite that offers a non-traditional interplay between automobiles and mobile devices. We have demonstrated that steering wheel controls can be seamlessly integrated into a variety of mobile applications and, along with speech recognition, can be used to create a new class of mobile applications. This class of applications gives drivers the ability to use their mobile devices while driving, in a manner that minimizes distractions. This is one of the key benefits of using data from a vehicle in real time. We were able to develop a set of constraints to determine whether it was applicable to present information to a driver. Similarly, we developed a design that makes it easy to add new constraints to our classifier, or to modify existing ones.

Our work demonstrates that it is possible to interleave vehicular controls for applications that deal with human-to-human interactions, as well as for those that deal with human-to-machine interactions. Similarly, we demonstrated that an existing application, in our case the Asgard City Browser, could be modified to accept steering wheel commands. Using steering wheel buttons allowed us to replace actions that would normally have to be carried out on the physical mobile device.

Because steering wheel space is limited, and because we did not want drivers to need to memorize an extensive list of button mappings across different applications, we made broad use of speech recognition in our applications. We used speech recognition to increase the types of commands an application would accept, without using up physical real estate on either the phone or in the car. The result was a more natural interaction between a user and an application; an application could pick up on specific commands or execute queries based on speech input. This frees the driver's eyes to focus on the main task at hand: driving.

With our design, we showed that it is possible to adapt OpenXC into a web-based application that preserves its original feature set, while simultaneously expanding the latter. We accomplished this in our CityBrowser application with a "compatibility layer." This compatibility layer did not require us to change any server-side code. Although our particular compatibility layer required injection of Javascript code, we have provided the abstraction of "patches" and "hijacks" to simplify this process.

We presented several designs for frameworks involving OpenXC-enabled applications, and determined that a decentralized organizational method would be best for our purposes. Our application suite employed this organizational framework, which made it simple to exchange data from one application to another. This allows us to provide seamless transitions between applications, for instance between CityBrowser and CarTalk. The goal of this is to leave the question of which application to use up to the framework, with the user left to concentrate on driving-related tasks.

In all, we hope to leave the reader with an analysis of a system that is readily extensible with the inclusion of additional applications, can monitor driver awareness in real time and use that awareness level to make decisions as to the presentation of content, and offers in-vehicle control options for both human-to-human and human-to-machine interactions.

6.2 Future Work

While our implementation serves as a working proof-of-concept, there are many additional components we hope to include in future work. CarTalk was designed with the abstraction of sharing "Shareable" objects, and we would like to extend the application to support more than text messaging. This includes adding support for sending audio and video snippets, as well as images.

In addition to extending the message types that CarTalk can send and receive, we want to extend the abilities of our awareness classifier. The goal here is to differentiate between different driving actions, and present the user with only the information or actions that they could absorb while in a particular state. For instance, city driving requires a different level of awareness than does highway driving. To quantify the states of awareness, user studies must be conducted. We hope to provide a "driver availability" classifier as a service in our application suite. This allows several applications to use the same criteria when determining driver awareness.

The OpenXC library, with the data that it allows to be gathered from a vehicle, has potential for a variety of applications. We have only demonstrated two distinct Android applications with our suite. There exists a myriad of possibilities, such as developing an application that shows drivers how to optimize their driving habits to get better gas mileage. We hope to extend our suite to include more applications, to make full use of the features that OpenXC provides.

Appendix

A1 Transcript of Example Usage of CarTalk and CityBrowser

Scenario: The user is driving a vehicle equipped with OpenXC, and has a mobile device with our application suite.

- The user launches the CityBrowser application on the mobile device and is presented with its mobile interface (Detailed in Figure 18).

- The user presses the steering wheel control button corresponding to "Query" (Detailed in Figure 17), and speaks the request "Find me restaurants in Boston that serve chicken parmesan." They release the "Query" button when they are done with their request.

- The list is read aloud to the user, and they touch an icon corresponding to the restaurant of their choice.

- The user presses the steering wheel control button corresponding to "Command" (Detailed in Figure 17) and speaks the request "Copy."

- The user presses the steering wheel control button corresponding to "Command" (Detailed in Figure 17) and speaks the request "Launch CarTalk." This launches the CarTalk application on the mobile device and brings the user to the "Send Message" interface (Detailed in Figure 15).

- The user presses the steering wheel control button corresponding to "Command" (Detailed in Figure 12) and speaks the request "To Jim," whereupon the "To:" field of the message is populated with "Jim."

- The user presses the steering wheel control button corresponding to "Command" (Detailed in Figure 12) and speaks the request "Hey Jim, I was wondering what you thought about this restaurant for dinner? CarTalk paste I feel like getting Italian food tonight." This request populates the "Body:" field of the message, and the phrase "CarTalk paste" is replaced with the contents of the the Android clipboard, which is the restaurant's information.

A2 Selected Code Examples

The SendMessage Activity

```
package com.openxc.apps;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.List;

import org.apache.http.HttpResponse;
import org.apache.http.HttpStatus;
import org.apache.http.client.ClientProtocolException;
import org.apache.http.client.HttpClient;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.impl.client.DefaultHttpClient;

import com.openxc.VehicleManager;
import com.openxc.measurements.Measurement;
import
com.openxc.measurements.UnrecognizedMeasurementTypeException;
import com.openxc.measurements.VehicleButtonEvent;
import com.openxc.measurements.VehicleButtonEvent.ButtonAction;
import com.openxc.measurements.VehicleButtonEvent.ButtonId;
import com.openxc.remote.VehicleServiceException;

import android.app.Activity;
import android.content.ClipboardManager;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
import android.content.pm.PackageManager;
import android.content.pm.ResolveInfo;
import android.os.AsyncTask;
import android.os.Bundle;
import android.os.Handler;
import android.os.IBinder;
import android.speech.RecognizerIntent;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;

public class SendMessage extends Activity {

     public static final String TAG =
"CarTalkClient:SendMessage";
     public static final String server =
"http://cartalkserver.appspot.com/";
     public static final SimpleDateFormat dateFormat = new
SimpleDateFormat("HH:mm MMM/dd");
     protected String user;
     private static final int REQUEST_CODE = 1111;

     TextView alerts;
     EditText toInput;
     EditText bodyInput;
     Button send;

     // OpenXC
    private VehicleManager mVehicleManager;
    private boolean mIsBound;
    private final Handler mHandler = new Handler();

    VehicleButtonEvent.Listener mButtonEvent =
            new VehicleButtonEvent.Listener() {
        public void receive(Measurement measurement) {
            final VehicleButtonEvent event =
(VehicleButtonEvent) measurement;
            mHandler.post(new Runnable() {
                public void run() {

checkVehicleButton(event.getValue().enumValue(),
event.getEvent().enumValue());
```

```
                }
            });
        }
    };

    private void checkVehicleButton(ButtonId buttonId,
ButtonAction buttonAction) {
      if (buttonId == ButtonId.OK) {
           if (buttonAction == ButtonAction.PRESSED) {
                startVoiceRecognition();
           }
      }
    }

    private ServiceConnection mConnection = new
ServiceConnection() {
        public void onServiceConnected(ComponentName className,
                IBinder service) {
            Log.i(TAG, "Bound to VehicleManager");
            mVehicleManager =
((VehicleManager.VehicleBinder)service
                    ).getService();

            try {

mVehicleManager.addListener(VehicleButtonEvent.class,
                mButtonEvent);
            } catch(VehicleServiceException e) {
                Log.w(TAG, "Couldn't add listeners for
measurements", e);
            } catch(UnrecognizedMeasurementTypeException e) {
                Log.w(TAG, "Couldn't add listeners for
measurements", e);
            }
            mIsBound = true;
        }

        public void onServiceDisconnected(ComponentName
className) {
            Log.w(TAG, "VehicleService disconnected
unexpectedly");
            mVehicleManager = null;
            mIsBound = false;
        }
    };

      @Override
      protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
      Bundle state = getIntent().getExtras();
      if (state != null && state.containsKey("user")) {
            user = state.getString("user");
            setContentView(R.layout.send);
            alerts = (TextView) findViewById(R.id.alerts);
            toInput = (EditText) findViewById(R.id.toInput);

            if (state.containsKey("to")) {
                  toInput.setText(state.getString("to"));
                  toInput.setEnabled(false);
            }

            bodyInput = (EditText) findViewById(R.id.bodyInput);
            send = (Button) findViewById(R.id.send);
            send.setOnClickListener(new OnClickListener() {

                        @Override
                        public void onClick(View v) {
                              String toInputString =
toInput.getText().toString();
                              String bodyInputString =
bodyInput.getText().toString();
                              if ((toInputString != "") &&
(bodyInputString != "")) {
                                    // Timestamp for the message
                                    String timestamp =
dateFormat.format(Calendar.getInstance().getTime());
                                    Send sendTask = new Send();
                                    // Sanitize the url before
sending it
                                    String command =
"from="+user+"&to="+toInputString+"&data="+bodyInputString+"&date
="+timestamp;
                                    command = command.replaceAll("
", "%20");
                                    command =
command.replaceAll("\\n", "%0A");
                                    sendTask.execute(command);
                                    alerts.setText("Sending ...");
```

```
                            } else {
                                alerts.setText("Message is
incomplete");
                            }
                        }
                    });

            // Speech to text
            Button speech = (Button) findViewById(R.id.speech);
            PackageManager manager = getPackageManager();
            // Check whether a recognizer is present
             List<ResolveInfo> activities =
manager.queryIntentActivities(
                    new
Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH), 0);
             if (activities.size() == 0)
             {
                 speech.setEnabled(false);
                 speech.setText("Recognizer not present");
             }

             speech.setOnClickListener(new OnClickListener() {

                        @Override
                        public void onClick(View v) {
                            startVoiceRecognition();
                        }
                    });
        }
        }

    @Override
    public void onResume() {
        super.onResume();

        bindService(new Intent(this, VehicleManager.class),
                mConnection, Context.BIND_AUTO_CREATE);
    }

    @Override
    public void onPause() {
        super.onPause();
        if(mIsBound) {
            Log.i(TAG, "Unbinding from vehicle service");
            unbindService(mConnection);
            mIsBound = false;
        }
    }

    private void startVoiceRecognition() {
        Intent intent = new
Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
        intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE_MODEL,
                RecognizerIntent.LANGUAGE_MODEL_FREE_FORM);
        intent.putExtra(RecognizerIntent.EXTRA_PROMPT, "CarTalk
Voice Recognition");
        startActivityForResult(intent, REQUEST_CODE);
    }

    @Override
    protected void onActivityResult(int requestCode, int
resultCode, Intent data) {
        if (requestCode == REQUEST_CODE && resultCode ==
RESULT_OK)
        {
            // Populate the wordsList with the String values the
recognition engine thought it heard
            ArrayList<String> matches =
data.getStringArrayListExtra(
                    RecognizerIntent.EXTRA_RESULTS);
            processVoiceResult(matches.get(0));
        }
        super.onActivityResult(requestCode, resultCode, data);
    }

    private void processVoiceResult(String match) {
        String pasteData = "";

        // If the speech was to set the TO: field
        String lowered = match.toLowerCase();
        if (lowered.startsWith("to")) {
            toInput.setText(lowered.replaceAll(" ",
"").replaceAll("to", ""));
            return;
        }

        // If this was a send message, perform a click on the
send button
```

```
            if (lowered.contains("car talk send")) {
                  send.performClick();
                  return;
            }

            ClipboardManager clipboard = (ClipboardManager)
getSystemService(Context.CLIPBOARD_SERVICE);
            if (clipboard.hasPrimaryClip()) {
                  pasteData = (String)
clipboard.getPrimaryClip().getItemAt(0).getText();
                  match = match.replaceAll("[Cc]ar( )*[Tt]alk(
)*[Pp]aste", pasteData);
            }

            bodyInput.setText(match);
      }

      public class Send extends AsyncTask<String, Void,
Boolean> {

            /**
             * returns whether sending the message was successful
             */
            @Override
            protected Boolean doInBackground(String... params) {
                  String data = params[0];
                  String result = "";
                  boolean retval = false;
                  HttpClient client = new DefaultHttpClient();
                  HttpPost post = new
HttpPost(server+"send?"+data);
                  try {
                        HttpResponse response =
client.execute(post);
                        if
(response.getStatusLine().getStatusCode() == HttpStatus.SC_OK) {
                              InputStream content =
response.getEntity().getContent();
                              BufferedReader buffer = new
BufferedReader(new InputStreamReader(content));
                              result = buffer.readLine();

                              retval = result.startsWith("Sent
message successfully");
```

```
                    }
                } catch (ClientProtocolException e) {
                    e.printStackTrace();
                } catch (IOException e) {
                    e.printStackTrace();
                }
                return retval;
            }

            protected void onPostExecute(Boolean result) {
                // anything that we do after sending a message
goes here

                if (result) {
                    alerts.setText("");
                    Intent intent = new
Intent(getApplicationContext(), Home.class);
                    intent.putExtra("user", user);
                    startActivity(intent);
                } else {
                    alerts.setText("Could not send message");
                }
            }
        }
}
```

The CarTalkMain Activity

```
package com.openxc.apps;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;

import org.apache.http.HttpResponse;
import org.apache.http.HttpStatus;
import org.apache.http.client.ClientProtocolException;
import org.apache.http.client.HttpClient;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.impl.client.DefaultHttpClient;

import android.app.Activity;
import android.content.Intent;
import android.os.AsyncTask;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;

public class CarTalkMain extends Activity {
     public static final String TAG = "CarTalkClient:Main";
     public static final String server =
"http://cartalkserver.appspot.com/";
     protected String user;
     protected String referrer;

     TextView alerts;
    EditText userEntry;
    Button loginButton;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        referrer = "";
        alerts = (TextView) findViewById(R.id.alerts);
        userEntry = (EditText) findViewById(R.id.username);
        loginButton = (Button) findViewById(R.id.login);
        loginButton.setOnClickListener(new OnClickListener() {

                @Override
                public void onClick(View v) {
                     Login loginTask = new Login();

      loginTask.execute(userEntry.getText().toString());
                     alerts.setText("Logging In...");
                }

        });
        alerts.setText("");

        // If we are to skip the login by being supplied the
username from another app
      Bundle state = getIntent().getExtras();
      if (state != null && state.containsKey("user")) {
           user = state.getString("user");

           if (state.containsKey("referrer")) {
                referrer = state.getString("referrer");
           }

           Login loginTask = new Login();
           loginTask.execute(user);
           userEntry.setText(user);
                alerts.setText("Logging In...");
      }
    }

    @Override
    public void onResume() {
      alerts.setText("");
      super.onResume();
    }

    public class Login extends AsyncTask<String, Void, String>
{
```

```
            @Override
            protected String doInBackground(String... params) {
                  String username = params[0];
                  String result = "";
                  HttpClient client = new DefaultHttpClient();
                  HttpPost post = new
HttpPost(server+"login?username="+username);
                  try {
                        HttpResponse response =
client.execute(post);
                        if
(response.getStatusLine().getStatusCode() == HttpStatus.SC_OK) {
                              InputStream content =
response.getEntity().getContent();
                              BufferedReader buffer = new
BufferedReader(new InputStreamReader(content));
                              result = buffer.readLine();

                              result =
(result.startsWith("SessionID:")) ? result.substring(10,
result.length()) : "";
                        }
                  } catch (ClientProtocolException e) {
                        e.printStackTrace();
                  } catch (IOException e) {
                        e.printStackTrace();
                  }
                  return result;
            }

            protected void onPostExecute(String result) {
                  // anything that we do after login goes here
                  user = (result == "") ? null : result;

                  if (user != null) {
                        Intent intent = new
Intent(getApplicationContext(), Home.class);
                        intent.putExtra("user", user);
                        if (referrer != "") {
                              intent.putExtra("referrer", referrer);
                        }
                        startActivity(intent);
                  } else {
                        alerts.setText("Login Failed");
                  }
              }

        }
    }
```

The OpenXC Browser

```
package com.quanta.nui.android.openxc;

import java.util.ArrayList;
import java.util.List;

import android.content.ClipData;
import android.content.ClipboardManager;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
import android.content.pm.PackageManager;
import android.content.pm.ResolveInfo;
import android.os.Bundle;
import android.os.Handler;
import android.os.IBinder;
import android.speech.RecognizerIntent;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.Toast;

import com.openxc.VehicleManager;
import com.openxc.measurements.Measurement;
import
com.openxc.measurements.UnrecognizedMeasurementTypeException;
import com.openxc.measurements.VehicleButtonEvent;
import com.openxc.measurements.VehicleButtonEvent.ButtonAction;
import com.openxc.measurements.VehicleButtonEvent.ButtonId;
import com.openxc.remote.VehicleServiceException;
import com.quanta.nui.android.openxc.R;
import com.quanta.nui.android.browser.WamiBrowser;

public class OpenXCBrowser extends WamiBrowser {

    public static final String TAG = "OpenXCBrowser";
    public ClipboardManager clipboard;
    private static boolean loaded;
```

```
        private String currentPOI;
        private Button button;
        private Button cartalk;
        private Button share;
        private static final int REQUEST_CODE = 1234;

        private static int delay;

        // Username
        private String user = "rkd";

        // OpenXC
    private VehicleManager mVehicleManager;
    private boolean mIsBound;
    private final Handler mHandler = new Handler();

    @Override
    protected String getInitialUrl() {
        return
"http://web.sls.csail.mit.edu/CityBrowser?mobile=true";
    }

    @Override
    protected int getContentViewId() {
        return R.layout.main;
    }

    private ServiceConnection mConnection = new
ServiceConnection() {
        public void onServiceConnected(ComponentName className,
                IBinder service) {
            Log.i(TAG, "Bound to VehicleManager");
            mVehicleManager =
((VehicleManager.VehicleBinder)service
                    ).getService();

            try {

mVehicleManager.addListener(VehicleButtonEvent.class,
                mButtonEvent);
            } catch(VehicleServiceException e) {
                Log.w(TAG, "Couldn't add listeners for
measurements", e);
            } catch(UnrecognizedMeasurementTypeException e) {
                Log.w(TAG, "Couldn't add listeners for
measurements", e);
            }
            mIsBound = true;
        }

        public void onServiceDisconnected(ComponentName
className) {
            Log.w(TAG, "VehicleService disconnected
unexpectedly");
            mVehicleManager = null;
            mIsBound = false;
        }
    };

    VehicleButtonEvent.Listener mButtonEvent =
            new VehicleButtonEvent.Listener() {
        public void receive(Measurement measurement) {
            final VehicleButtonEvent event =
(VehicleButtonEvent) measurement;
            mHandler.post(new Runnable() {
                public void run() {

checkVehicleButton(event.getValue().enumValue(),
event.getEvent().enumValue());
                }
            });
        }
    };

    private void checkVehicleButton(ButtonId buttonId,
ButtonAction buttonAction) {
      if (buttonId == ButtonId.UP) {
            if (buttonAction == ButtonAction.PRESSED) {
                  clickMicrophone();
            }
      } else if (buttonId == ButtonId.DOWN) {
            if (buttonAction == ButtonAction.PRESSED) {
                  startVoiceRecognition();
            }
      }
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mWebView.addJavascriptInterface(new HijackJSInterface(),
"hijack");
        clipboard = (ClipboardManager)
getSystemService(Context.CLIPBOARD_SERVICE);
        currentPOI = "";
        loaded = false;
        Log.i(TAG, "OpenXCBrowser created");
        button = (Button) findViewById(R.id.button);
        button.setOnClickListener(new OnClickListener() {

                @Override
                public void onClick(View v) {
                     clickMicrophone();
                }

        });
        cartalk = (Button) findViewById(R.id.cartalk);
        cartalk.setOnClickListener(new OnClickListener() {

                @Override
                public void onClick(View v) {
                     launchCarTalk();
                }
           });
        share = (Button) findViewById(R.id.share);
        share.setOnClickListener(new OnClickListener() {

                @Override
                public void onClick(View v) {
                     copyPOIToClipboard();
                }
           });
           share.setClickable(false);
           share.setActivated(false);

           // Speech to text
           Button speech = (Button) findViewById(R.id.speech);
           PackageManager manager = getPackageManager();
           // Check whether a recognizer is present
        List<ResolveInfo> activities =
manager.queryIntentActivities(
                new
Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH), 0);
        if (activities.size() == 0)
        {
           speech.setEnabled(false);
            speech.setText("Recognizer not present");
        }

        speech.setOnClickListener(new OnClickListener() {

                @Override
                public void onClick(View v) {
                     startVoiceRecognition();
                }
           });

        // Make the buttons invisible
        button.setVisibility(View.GONE);
        cartalk.setVisibility(View.GONE);
        speech.setVisibility(View.GONE);
        share.setVisibility(View.GONE);
    }

    private void copyPOIToClipboard() {
           ClipData clip =
ClipData.newPlainText("CityBrowserPOI", currentPOI);
           clipboard.setPrimaryClip(clip);
           Toast toast = Toast.makeText(getApplicationContext(),
"Copied to Clipboard", Toast.LENGTH_SHORT);
           toast.show();
    }

    private void launchCarTalk() {
           // Launch cartalk and skip the login sequence
           PackageManager manager = getPackageManager();
           Intent intent = new Intent();
           intent =
manager.getLaunchIntentForPackage("com.openxc.apps");
           intent.addCategory(Intent.CATEGORY_LAUNCHER);
           intent.putExtra("user", user);
           intent.putExtra("referrer", "citybrowser");
           startActivity(intent);
    }

     private void startVoiceRecognition() {
        Intent intent = new
Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
        intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE_MODEL,
                RecognizerIntent.LANGUAGE_MODEL_FREE_FORM);
        intent.putExtra(RecognizerIntent.EXTRA_PROMPT,
"CityBrowser Voice Recognition");
        startActivityForResult(intent, REQUEST_CODE);
    }

      @Override
      protected void onActivityResult(int requestCode, int
resultCode, Intent data) {
        if (requestCode == REQUEST_CODE && resultCode ==
RESULT_OK)
        {
            // Populate the wordsList with the String values the
recognition engine thought it heard
            ArrayList<String> matches =
data.getStringArrayListExtra(
                    RecognizerIntent.EXTRA_RESULTS);
            processVoiceResult(matches.get(0));
        }
        super.onActivityResult(requestCode, resultCode, data);
      }

      private void processVoiceResult(String match) {
           match = match.toLowerCase();
           if (match.contains("share") || match.contains("copy"))
{
                copyPOIToClipboard();
           } else if (match.contains("car talk")) {
                launchCarTalk();
           }
      }

    @Override
    public void onResume() {
      super.onResume();
        bindService(new Intent(this, VehicleManager.class),
                mConnection, Context.BIND_AUTO_CREATE);
        Log.i(TAG, "OpenXCBrowser resumed");
    }

    @Override
    public void onPause() {
      super.onPause();
        if(mIsBound) {
            Log.i(TAG, "Unbinding from vehicle service");
            unbindService(mConnection);
            mIsBound = false;
        }
    }

    // Simulates a user's click to begin recording
    private void clickMicrophone() {
      loadUrl("javascript: $('#wami-microphone > div > div >
div').click();");
      currentPOI = "";
      if (!loaded) {
          hijackAddMarker();
          loaded = true;
      }
    }

    // Hijacks the addMarker function
    private void hijackAddMarker() {
      String sled = "javascript: var originalAddMarker =
addMarker; function addMarker(location, contentString, imgUrl,
title, _oid) {" +
                "     var marker = new google.maps.Marker({"+
                "     position : location," +
                "     map : map," +
                "     icon : imgUrl," +
                "     title : title," +
                "     optimized : false" +
                "});" +
                "marker._oid = _oid;" +
                " " +
                "google.maps.event.addListener(marker, 'click',
function () {" +
                "     infowindow.setContent(contentString);" +
                "     infowindow.open(map, marker);" +
                "     fireCityEvent('MARKER_CLICKED', marker);" +
                "     hijack.sendToAndroid(contentString);" +
                "});" +
                " " +
                "markersArray.push(marker);" +
                "}";
        loadUrl(sled);
    }

    public class HijackJSInterface {
     public HijackJSInterface() {
     }

     public void sendToAndroid(String text) {
           text = text.replaceAll("<br>", " \n ");
           text = text.replaceAll("</(\\w)+>", " \n ");
           text = text.replaceAll("<(\\w)+>", "");
           text = text.replaceAll("&bull;", ",");
           currentPOI = text;
           share.setClickable(true);
           share.setActivated(true);
           Log.v(TAG,text);
     }
    }
}
```

References

[1] Nielson. Survey: New U.S. Smartphone Growth by Age and Income, February 20, 2012. http://blog.nielsen.com/nielsenwire.

[2] Tabletpcsplayer.com, April 13, 2012.

[3] D. Williams. The Arbitron National In-Car Study. Arbitron, Inc. 2009.

[4] Polk. Average Age of Vehicles Reaches Record High, January 17, 2012. http://www.polk.com.

[5] M. Richtel. Consumers Hold onto Products Longer. *The New York Times,* February 25, 2011.

[6] http://openxcplatform.com/

[7] Bosch. Road vehicles - Controller area network (CAN) - Part 1: Data link layer and physical signaling. ISO 11898-1:2003 (ISO) 2003.

[8] A. Gruenstein, J. Orszulak, S. Liu, S. Roberts, J. Zabel, B. Reimer, B. Mehler, S. Sene, J. Glass, and J. Coughlin. City Browser: developing a conversationalautomotive HMI. CHI, 2009.

[9] A. Gruenstein, I. McGraw, and I. Badr. The WAMI toolkit for developing, deploying, and evaluating web-accessible multimodal interfaces. ICMI, 2008.