Addressing the Challenges of HTTP-based Mobile/Cloud Interaction

Jules White, Yao Pan, Zach McCormick
Vanderbilt University
{jules.white, yao.pan, zach.mccormick}@vanderbilt.edu

November 5, 2013

***Abstract*—A new trend is the construction of mobile cyber-physical applications that rely on sensor measurements from multiple mobile devices and back-end processing power in the cloud. Most of these applications use HTTP for network communication, which adds significant data marshaling, threading, synchronization, and other complexities to the device-cloud interactions. This paper presents an architectural approach for using distributed object computing concepts and web sockets to address the key challenges of simultaneously supporting legacy HTTP-based infrastructure and providing simplified mobile/cloud interactions.**

***Keywords*-mobile computing, cloud computing, distributed object computing**

I. Introduction

Emerging trends and challenges. An emerging trend in computing is to build applications with complex cyber-physical behavior, where the application collects, processes, and reacts to information from the physical world. Mobile devices are key drivers of these new cyber-physical applications, since they contain a variety of sensors, such as GPS, cameras, and accelerometers, coupled with easy-to-use, high-level APIs for using them. For example, researchers have built applications to coordinate and manage disaster rescue [1], [2], detect traffic accidents and provide situational awareness service to first responders [3], track and analyze CO2 emissions [4], facilitate access of cyber-information in construction sites [5], monitor physical and mental health condition and provide community-guided feedback [6], and detect real-time traffic information to forecast the fastest routes for taxis [7].

A common challenge with these mobile cyber-physical systems is that they require either: 1) processing or aggregating large volumes of information from multiple devices; 2) applying computationally intensive algorithms to the data; or 3) tight coordination of multiple mobile devices to achieve a goal. For example, in Thiagarajan's system [7] mentioned above, there are thousands of drivers on the road, each with mobile devices, and large volumes of GPS data are aggregated and analyzed in real-time for route recommendation. In Bae's application [5] of mobile augmented reality, computationally intensive computer vision algorithms are used to derive and present relevant cyber-information about the physical world, and it is impractical to do these computations on a mobile device. Because of the limited processing power, finite battery life, and intermittent connectivity on mobile devices, building these applications solely on mobile platforms is hard.

Connecting mobile devices to cloud computing resources is the most common approach to overcoming the data aggregation, processing, and coordination challenges of building these mobile cyber-physical systems. A cloud-based back-end server is used to aggregate results from one or more devices, apply substantial back- end processing to the data, and then disseminate results and coordination actions to mobile devices. These same cloud back-ends typically support browser-based clients as well, with one example being Facebook's browser-based site and mobile app.

A variety of forces have driven Hyper Text Transfer Protocol (HTTP) to be the dominant network communication protocol for device-cloud coordination. For many organizations, their infrastructure is based on existing systems using HTTP, thus allowing mobile cyber-physical apps to use the same remote services APIs and programming language abstractions that are already used by an organization's browser-based applications. Moreover, organizations can reuse the substantial number of off-the-shelf HTTP load balancing, caching, and session frameworks that are available. More advanced distributed communication protocols, such as CORBA, are rarely used for device-cloud interaction because they do not readily provide the capability for organizations to use a single consistent back-end interface and infrastructure for both browser and mobile clients.

Open problem. Despite the apparent software and infrastructure reuse advantages of leveraging an HTTP-based backend for device-cloud coordination, there are a number of significant disadvantages due to design decisions inherent in the HTTP protocol. One of the most significant challenges is that HTTP is primarily a client/server protocol and does not support server-initiated communication – although newer web socket standards are beginning to change this model. The key complexities introduced by HTTP include: 1) the difficulty of using standard software design patterns, such as the Observer Pattern, for device-cloud interactions due to the non-object-oriented nature of the HTTP protocol; 2) the difficulty of supporting 2-way communication between mobile devices and the cloud; and 3) the difficulty of supporting direct discovery of, and interaction with, mobile devices in the same physical context, such as those in the same room. These mismatches between the design of HTTP and the requirements needed for tight device-cloud interaction in mobile cyber-physical systems has created a gap in research for new approaches that can support a common back-end infrastructure and protocol for both browsers and mobile clients that facilitates the reuse of existing HTTP-based infrastructure, such as load balancers, and also can address each of the four challenges outlined above.

Solution approach ⇒ Analyzing & Addressing the Architectural Deficiencies of HTTP-based Mobile/Cloud Interaction. To address the key challenges laid out above, we carefully analyzed the architectural issues inherent in HTTP and push messaging that lead to the above challenges in device-cloud interaction for cyber-physical systems. This paper presents specific examples of how HTTP-based architectures could be improved to eliminate these challenges. The architectural improvements are presented in the context of SodaCloud, a reference implementation of the proposed architectural improvements. SodaCloud uses a combination of transparent object reference sharing, web socket-based communication, and contextual object discovery to overcome these challenges of building mobile cyber-physical applications that interact with the cloud. Furthermore, it provides a common interface for both browsers and mobile clients while maintaining compatibility with existing HTTP infrastructure.

The remainder of this paper is organized as follows: Section II discusses key challenges and architectural issues of using HTTP and push messaging for device-cloud interaction; Section III discusses our proposed architectural improvements and reference implementation; Section V presents empirical results from experiments with our reference implementation; Section V compares SodaCloud with related work; and Section VI presents concluding remarks and lessons learned.

II. Challenges

Challenge 1: Two-way Interaction With The Cloud is Challenging. Creating applications with tightly coordinated 2-way device-cloud interaction is challenging. Push messaging, often built on top of alternate protocols such as XMPP, has emerged as a stop gap, but is still not ideal since it introduces new complexities. Applications using push messaging must develop or use yet another protocol, deal with complex synchronization/threading complications, and be prepared for a host of network connectivity issues.

Challenge 2: Implementing Common Software Design Patterns is Difficult. Substantial complexity is introduced when transitioning the boundary between object-oriented (OO) programming abstractions, such as method calls and objects, and the sequential request-response abstraction of HTTP, which makes the use of common software design patterns hard. Many device-cloud systems exhibit requirements that could be easily addressed with well-known software design patterns, such as the *Observer Pattern*. For example, a mobile client may need to be notified whenever a specific data structure in the cloud changes state, which would motivate the use of the Observer pattern. However, the HTTP protocol eliminates all concepts of objects, references, and method calls and makes it hard to register a mobile client's listener object to receive callbacks from a server-side data structure encapsulated in a cloud object.

Challenge 3: Lack of Support for Discovery of Cyber-identity Based on Context. Cyber-physical peer-to-peer discovery, addressing, and interaction are hard to implement on top of HTTP. This is problematic for many cyber-physical applications, which require these features to obtain knowledge of contextual information for accurate coordination of cyber-physical interaction. For example, a doctor may want to send a medical record from her tablet to the mobile phone of the nurse standing in front of her. HTTP and push messaging do not provide support for discovery of a cyber-identity based on the physical context of a device (*e.g.*, discovering the address of a person standing in front of me in order to send them a medical record). Moreover, mediating these connections over HTTP through a cloud server requires developing custom messaging solutions to queue up and deliver messages to interaction participants across client-driven communications.

A. Architectural Analysis of Challenges

To understand the architectural sources of the three challenges above, we carefully analyzed a variety of cyber-physical mobile applications built on top of both HTTP polling and HTTP + push messaging. Our goal was to identify the fundamental design forces at work in these architectures that produced the challenges we outlined. From our analysis of cyber-physical mobile applications that both we and others have written, we expounded two key architectural decisions creating the challenges that we have outlined:

1) Architectural Issue 1: Push messaging delivery and addressing stops at the app boundary and does not support intra-app object/method addressing and delivery.
2) Architectural Issue 2: The standard push architectures do not directly support exchange and contextual discovery of object references or peer-to-peer communication across devices.

The lack of intra-app object/method addressing and delivery, requires developers to write custom code for determining the appropriate recipient object and method to execute on that object. Moreover, the custom code must unmarshall the message into a form that the recipient object can accept. Delivery of the final message to the target object may also require synchronization between one or more network threads and a display thread that is rendering the GUI.

For example, on Android, there are two common approaches for implementing two-way device-cloud communication. The first approach is to use HTTP to send messages to the cloud and push notifications to receive cloud-initiated communication. The second approach is to use HTTP polling, which requires the mobile client to continually send requests to the cloud in order to receive any messages that the cloud has queued up for it. In both scenarios, the lack of intra-app object messaging forces the developer to cross a non-object-oriented communication

boundary, which includes significant marshalling, threading, and message routing complexity. For example, when an application receives a push message, it must place that message into a message queue used by the display thread in order to update the GUI without causing a threading issue. Similarly, on Android, an application that uses HTTP polling to send and receive messages will need to send and receive those messages in a separate thread that must exchange the messages with the main GUI thread upon receipt. Moreover, in both situations, the application will require custom marshalling logic to convert the message contents from either a raw HTTP response or push message to an object-oriented message.

To prevent blocking on the UI-handling thread, all network operations are usually performed in a separate thread. In Android, this can be accomplished in a variety of ways, including Java Threads and Android-specific AsyncTasks, as shown below:

```
public class AddReportTask extends AsyncTask<
    Void, Void, Void> {
...
  @Override
  protected Void doInBackground(Void...
      params) {
    HttpClient httpclient = new
        DefaultHttpClient();
    HttpPost request = new HttpPost(Host.
        hostaddress + "addreport/");
      try {
        request.setEntity(myEntity);

        HttpResponse response = httpclient.
            execute(request);

        if (response.getStatusLine().
            getStatusCode() == HttpStatus.
            SC_OK) {
          // do something with the response
              ...
        }
      } catch (IOException e) {...}
    return null;
  }
...
```

Listing 1: Client to Server Invocation

Clearly, there are significant development complexities in the above example, especially considering more than one type of object (thus more than one type of AsyncTask). Moreover, it is important to point out that this example does not include any form of polling or push messaging, which further increases the threading, message routing, and synchronization complexity.

One of our fundamental assertions is that these layers of complexity can be eliminated by simply introducing intra-app object/method addressing. With an intra-app object/method addressing approach, client-side objects can use well-known distributed object computing approaches to easily invoke methods on specific remote objects living in the cloud or on the mobile client. A number of existing distributed object computing approaches, such as CORBA, provide intra-application object/method addressing, however, they have not gained widespread usage on mobile platforms, typically due to their significant overall complexity and lack of support for browser-based clients or existing HTTP infrastructure. Some approaches, such as Google App Engine's Cloud Endpoints, provide uni-directional RPC-based object-oriented communication to cloud services. However, these existing approaches do not completely solve Architectural Issue 1, since they only provide device-to-cloud intra-application object/method addressing and delivery. They do not support bi-directional object-oriented communication. The majority of common software design patterns include object interactions where multiple objects need direct bi-directional communication with each other in order to function properly. To illustrate this concept, we will use the Observer pattern again. Let's assume that we have a ReportManager object that is participating in the Observer pattern:

```
public class ReportManagerImpl implements
    ReportManager {
  private List<ReportListener> listeners_
      = ...;

  public void addListener(ReportListener l
      ){listeners_.add(l);}

  public void addReport(Report r){
    ....//add the report
    for(ReportListener l : listeners_){
      l.reportAdded(r); //notify devices
          of new report
    }
  }
}
```

Typically, we would expect the ReportManager object to be hosted in the cloud and manage a list of reports stored in a database. Without cloud-to-device intra-app object/method addressing, there is no way with existing approaches for the cloud to notify a listener object on a device when a report is added. The device-to-cloud intra-application object/method addressing and message delivery would allow the device to send its listener to the `addListener` method of the ReportsManager. However, the ReportsManager has no communication pathway back to the device to invoice the `reportAdded` method on the device's listener within the `addReport` method.

III. Solution Approach ⇒ Intra-app Object/Method Messaging & Contextual Object Reference Discovery

Based on our analysis of the architectural issues that add complexity to building mobile cyber-physical appli-

cations with cloud backends, we set out to create, implement, and experiment with a reference implementation of an architecture that fixed the key issues we found. To effectively demonstrate our solution approach, we built a reference implementation of the proposed architecture for JavaScript, Java, and Objective-C, allowing us to experiment with Android, iOS, and browser platforms simultaneously. Our open-source reference implementation is called SodaCloud and is available for download from (https://github.com/VT-Magnum-Research/sodacloud). An overview of the architecture is shown in Figure 1. This section describes each component in detail, as well as some specific implementation issues in each of the target languages.

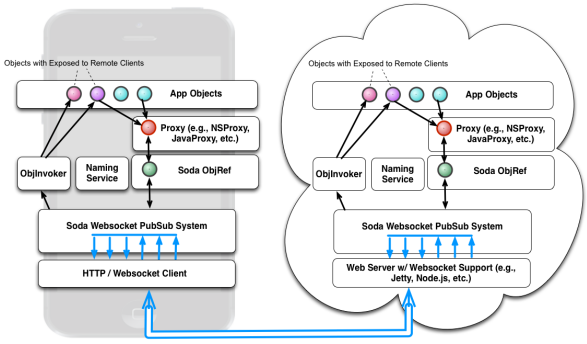

Figure 1: SodaCloud Architecture

A. Intra-app Object/Method Addressing via Object Proxies

A core concept of SodaCloud is that it uses fully object-oriented programming abstractions for all client-to-cloud and client-to-client interactions. In order to provide an object-oriented interface for communication, SodaCloud uses proxies to hide the messaging, threading, synchronization, and marshalling intricacies of method calls. The code shown below imitates the functionality in Listing 1, but uses SodaCloud for communication:

```
// Soda is our implementation of the
    research approach
ReportManager mgr = soda.get("reportMgrId",
     ReportManager.class);

Report r = new Report(...);
mgr.addReport(r);
```

Listing 2: Equivalent of Listing 1 Using Intra-app Object/Method Addressing & Delivery

The key to hiding these complexities is the use of proxy objects to translate local method calls on a remote reference to an object into a series of messages sent either to an object belonging to the cloud or a client, as shown in Figure 1. Remote object references are obtained by looking up a reference to the object through either a name bound to SodaCloud's naming service or via a direct object reference string obtained through contextual discovery (*e.g.,* scanning an object reference from a barcode). The example in Listing 2 shows the use of SodaCloud's naming service to obtain a reference to a remote object of type `ReportManager`. Rather than returning an actual instance of a ReportManager object, SodaCloud instead returns a proxy object that translates method calls into messages sent to the actual remote object on another host. This setup mirrors approaches that are seen in frameworks such as CORBA.

The underlying communication mechanism, as shown in Figure 1, is based on a publisher/subscriber network built on top of web sockets. Web sockets, defined by RFC 6455 [8], provide an extension to the HTTP protocol for bi-directional socket-based binary communication. A set of cloud-based servers form the backbone of the publisher/-subscriber network and move messages between servers and mobile clients. Each host in the publisher/subscriber network listens for messages on a unique topic that is assigned to it. Topics allow a single client to subscribe to the topic and can require authentication (*e.g,* to ensure that messages are not intercepted).

The rationale for using web sockets as the core transport is that they operate on top of HTTP. Up-to-date HTTP-based server and load balancing infrastructure can be used, since the initial establishment of a connection is based on HTTP. No port changes are needed since the protocol operates on top of HTTP. Furthermore, current browsers support the web socket protocol, allowing both browsers and mobile clients to share a single common communication protocol that supports bi-directional communication.

Each object that is remotely accessible is assigned a unique address. The address is composed of two parts: 1) the topic ID of the host that the object resides on and 2) a UUID that uniquely identifies that specific object on the host. When a host receives an incoming message from its associated topic, it determines the correct recipient object by looking at the address the message is addressed to, stripping out the host topic ID, and using the UUID to obtain the target object from a map.

B. Publisher/Subscriber Intra-app Object/Method Invocation

The primary method of obtaining object references is through method invocations on proxies. By default, when a mobile client connects to a cloud host, it automatically creates a proxy to the naming service on the cloud host. Cloud hosts typically bind their naming services to a special root object address. Other naming services cannot be bound to this address.

When a method is invoked on a proxy, the proxy determines which of the argument values passed to the method should be passed by value and which should be passed by reference. Argument values passed by reference are automatically converted into a specialized format which

includes the address of the object on the host. All other argument values are converted to Javascript Object Notation (JSON) representations. The decisions of what objects to pass by reference vs. value are driven by the programmer and are controlled by language-specific features (*e.g.,* Java Annotations, etc.).

Once a method call is marshaled by the proxy into an invocation message, the proxy publishes the message to the topic of the host that the remote object resides on. The publisher/subscriber network takes care of delivery of the message to the target host, regardless of whether or not the host is a cloud server or another mobile client. The publisher/subscriber network also manages queueing of messages. Clients can specify max queue times or other hints to help the publisher/subscriber network determine if or how long a message should be queued if it cannot immediately be delivered.

When the target host receives an invocation message, its local `ObjInvoker` handles the determination of the target object and the demarshalling of the method call parameters. The `ObjInvoker` then invokes the target method on the specified object, determines the return value (if any), and publishes a reply message to the topic of the sending host that specifies the return value or thrown exception of the invocation. The proxy object that sent the invocation automatically demarshalls the result and either provides the return value as a result of the call or throws the exception that was included in the reply message. If the return value is an object reference, the proxy automatically converts the object reference into a new proxy to the remote object on the remote host.

```
// Soda is our implementation of the
    research approach
ReportManager mgr = soda.get("reportMgrId",
     ReportManager.class);
mgr.getReports(new ReportCallback(){
   public void reportCallback(Report[]
       reports){
       //do something with the async
           callback value
   }
});
```

Listing 3: Passing an Async-callback via Object Reference

This transparent creation of proxies is a critical capability for maintaining object-oriented abstractions and supporting intra-app object/method addressing. For example, standard object-oriented software design approaches for retrieving the value of long-running computations can be used. As shown in Listing 3, a callback object can be provided as a parameter of the method invocation and passed by reference to the remote method invocation. The remote object, as shown in Listing 4, upon completing the invocation, can then use the passed object reference (which is remotely unmarshalled as a proxy) to invoke the callback function and provide the return value.

```
public class ReportManagerImpl implements
    ReportManager {
    ...
    public void getReports(ReportCallback
        callback){
        ...
        callback.reportCallback(this.
            myReports);
    }
}
```

Listing 4: Server-side Usage of Client-provided Callback

A powerful capability of this transparent object reference passing paradigm is that the remote object can treat the passed callback as if it were a local object, regardless of the actual language of the remote callback object. For example, as shown in Listing 5, a browser-based Javascript client can pass a Javascript object as a callback to the Java `ReportManagerImpl` object shown in Listing 4. On the `ReportManagerImpl`'s host, the Javascript callback is demarshalled as a Java proxy to a remote object. When the `ReportManagerImpl` invokes the `reportCallback` method, the method invocation is translated by the proxy into a message, published to the topic of the browser hosting the Javascript object, and then delivered to the target Javascript object by its local `ObjInvoker`.

```
Soda.get("reportMgrId", function(mgr){
  var callback = {
     reportCallback:function(reports){
         //do something with the async
             callback value
     }
  };
  mgr.getReports(callback);
});
```

Listing 5: Javascript Passing an Async-callback via Object Reference

Supporting this type of complex 2-way object interaction with existing HTTP polling or HTTP polling + push approaches is extremely challenging. In our architectural analysis, we did not find a single app that supported these types of complex interactions. Moreover, we found no existing frameworks that can support transparent passing of object references from mobile device to cloud (and vice-versa) over HTTP.

C. Automatic Threading & Synchronization

An important aspect of maintaining the object-oriented interface that proxy objects provide is ensuring that method invocations provide the appropriate synchronization and blocking semantics to mirror a standard local method invocation. All method invocations on a proxy that return a value are automatically synchronous. The proxy sends the

invocation message to the topic of the host with the remote object and then blocks until it receives reply or times out.

Void methods can each individually be configured to either provide synchronous execution or asynchronous execution. Language specific features in each implantation are used to designate synchronous versus asynchronous void methods. For example, in Java, SodaCloud uses the `@SodaSync` annotation to designate void methods that should be called synchronously. For asynchronously invoked void methods, marked `@SodaAsync` in Java, control is immediately returned to the calling thread. It is guaranteed that asynchronous void invocations always occur in the same order on the remote host as they occurred on the host of the caller. Therefore, if a caller invokes two asynchronous void methods in series, the remote object will receive those invocations in the same sequential order.

SodaCloud can also be extended with platform specific annotations to aid in threading and synchronization. For example, on Android, the `@SodaInvokeInUI` annotation can be used to designate methods that should always be invoked in the main/UI thread. The SodaCloud Android implementation automatically handles transferring the call through an Android `Handler` to the UI thread for invocation. This concept is particularly powerful for implementing common UI software design patterns, such as the Model View Controller (MVC) pattern. In the code shown in Listing 6, an MVC controller registers a listener on a remote model object residing in the cloud and automatically receives invocations from the cloud in the context of the UI thread so that it can automatically update the views. The developer only needs to specify the intended target threading context for SodaCloud to handle the underlying complexities of threading and synchronization.

```
...
mgr.addReportsListener(new ReportListener()
    {
   @SodaInvokeInUI
   public void reportsChanged(Report[]
       reports){
       updateReportListView(reports);
   }
});
```

Listing 6: Passing an Async-callback via Object Reference

D. Contextual Object Reference Discovery

SodaCloud includes facilities for sharing and discovering objects using physical context. For example, object references can be stored in QR codes and then scanned onto another device in order to allow it to create a proxy to the specified object. In our open-source SodaCloud reference implementation, we include an example application that uses SodaCloud to forward the text messages received by an Android device to a user's browser (https://github.com/juleswhite/cs27x/tree/master/assignments/Asgn1). The browser can also compose and send text messages through the Android device. In order to connect the mobile device to the browser, the mobile device needs to discover an object reference of the controller for the UI drawn in the user's browser. In order discover this object, the browser creates the controller object and publishes an object reference for it as a QR code that is displayed on screen, as shown in Listing 7. The mobile device then scans the reference to the JavaScript object embedded in the QR code and creates a proxy to the controller object in order to forward it text messages as they are received. The mobile device also calls the `smsSenderAdded` method on the Javascript controller object and passes it a reference to an Android object that the browser can use to send outgoing text messages.

```
var uri = Soda.namingSvc.publish({
  smsEvent:function(evt){
      ...
  },
  smsSenderAdded:function(sender){
       ...
  }
});
var encodedObjRef = encodeURIComponent(uri);
...
<div>
   Scan this QR code to connect:
   <img src="http://chart.apis.google.com/
       chart?cht=qr&chs=350x350&chld=L&choe=
       UTF-8&{{encodedObjRef}}"></img>
</div>
```

Listing 7: Embedding an Object Reference in a QR Code

Creating this type of complex browser to mobile device interaction is extremely difficult with existing HTTP polling and HTTP + push approaches. Moreover, there are no distributed object computing frameworks for both mobile and browser-based applications, which we are aware of, that can support this type of complex object-oriented interaction and contextual object discovery.

SodaCloud also supports attaching object references to specific locations so that they can be discovered by other clients. For example, in the code shown in Listing 8, SodaCloud is used to bind an object reference to an area with a 20 meter radius surrounding a specific latitude and longitude. A maintenance app can then discover report objects related to specific pieces of equipment in the area by simply looking up objects within a specific proximity of the device.

```
Soda soda = new Soda();
MaintenanceReport r = ....//create a report
    for a piece of equipment
soda.bind(r).to(SodaLocation.within(Proximity
    .TWENTY_METERS).of(37.2, -80.4).
    atAccuracy(Accuracy.FINE));
List<Runnable> l = soda.find(Runnable.class,
    SodaLocation.at(37.2, -80.4)).now()
```

Listing 8: Attaching an Object Reference to a Physical Location

Object references can easily be shared because they can include both an access point to the publisher/subscriber network as well as the host topic and UUID of the object that they represent. Discoverers that are not already connected to the publisher/subscriber network can then immediately connect to a publisher/subscriber network access point and begin interacting with the specified object. Other ways that object references could be shared are through Near-Field Communication (NFC), Bluetooth, Wi-Fi, or acoustic signals, such as ultrasound.

IV. Comparative Analysis of Mobile/Cloud Interaction Architectures

To compare the overall advantage and disadvantages of each of the mobile client to cloud interaction architectures, we devised a series of analyses. The analyses look at a range of architectural aspects, such as threading complexity, synchronization complexity, communication latency, and message overhead. The results of these analyses are presented throughout the remainder of this section.

A. Analysis 1: Communication Performance & Overhead Evaluation

The first analysis focused on the overall performance of the communication architecture itself. We analyzed a common scenario in mobile / cloud cyber-physical systems whereby a mobile client initiates a long-running or other operation in the cloud and is asynchronously notified by the cloud of the result at a later point in time. This setup is common in a variety of mobile cyber-physical applications, such as:

- initiating server-side image processing and being asynchronously notified of the results
- registering for aggregate analysis of sensor inputs from multiple devices and being notified when new aggregate results are available
- listening to changes in one or more other mobile client's locations
- submitting a Map/Reduce job for processing in the cloud

Analysis Design. The overall goal was to understand: 1) the latency between the start of the interaction from the mobile client and the asynchronous delivery of the result to the client from the cloud; 2) the overall messaging overhead incurred by the interactions; 3) the maximum rate of interaction (*e.g.,* for high-throughput interactions); 4) the maximum message sizes that can be exchanged (*e.g.,* which is important when transmitting physical sensor measurements, such as imagery); and 5) whether or not the interactions supported browser-based clients.

Rather than picking a single specific implementation of a HTTP polling or push messaging, we instead focused on the common generic architecture that each of these approaches exhibits regardless of the implementation used. An important reason that we chose to analyze the architectures themselves rather than specific implementations is that we did not want variations in implementations or configurations to be assessed. Instead, we wanted to focus on the abstract invariant properties of the architectures. For example, rather than measuring latency in milliseconds, we instead measure latency in terms of the number of messaging hops required to complete the interaction. Rather than measuring exact sizes of messages, we instead focus on the invariant messaging overhead incurred by the underlying protocols used in the architecture.

Analysis 1 Results. The overall results of the analysis are shown in Table I. The top of Table I shows the overall messaging interactions required to complete the scenario and servers involved. For SodaCloud, a single bi-directional web socket is used to exchange messages. HTTP Polling, as shown in the table, sends periodic messages to the server in order to receive updates. The first HTTP+Push architecture exhibits the most common structure for push messaging, whereby the cloud application delivers the asynchronous result to the client by sending an HTTP request to a push server asking it to push the result to the mobile client. The push server then manages the sending of the push message to the mobile client. Most push messages are limited in size by the push messaging provider, so we have included a fourth architecture where the push message triggers the client to poll the server and receive a larger message that would not fit into a push notification.

The first analysis of importance is the round-trip latency to complete the entire interaction. We measured the overall latency and terms of the number of messaging hops through participants in the architecture. Because SodaCloud is based on direct web socket connection to the cloud, only a single hop is required. HTTP Polling requires sending the initial request to the server and then some number of additional requests in order to retrieve the result. The overall latency is at least one top plus the polling interval. The push architectures both require interaction with an auxiliary push messaging server, which introduces 1 to 2 additional hops in the architecture. Moreover, push messaging providers in force rate limitations on interactions and may queue and delay messages. the overall latency of these architectures is between 3 to 4 hops plus any queuing delay imposed by push messaging provider.

Another important aspect of the architectures is the rate at which interactions can occur and the sizes of the messages that can be exchanged. For example, in sensor driven mobile cyber physical applications, mobile devices and cloud infrastructure may need to support the transfer of imagery/video or bursty periods where large numbers of messages are exchanged at a high rate. the rate of message exchange and soda and the size of messages that can be exchanged are not artificially bounded by the architecture itself. HTTP polling has no bounds on message size, however, the rate of

	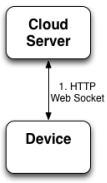	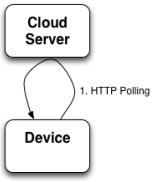	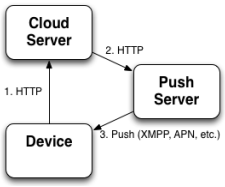	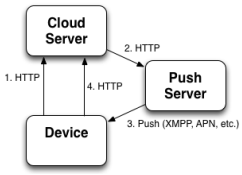
	SodaCloud	HTTP Polling	HTTP + Push	HTTP + Push (large msgs)
Max Latency	1-hop	1-hop+Polling Interval	3-hops + queuing	4-hops + queuing
Max Cloud to Device Invocation Rate	No limit	Polling Rate	Rate-limited by Provider	Rate-limited by Provider
Max Message Size	No Limit	No Limit	256 bytes - 4K	No Limit
Per Invocation Message Overhead	2 bytes	200 bytes X #of polls	(200 bytes X 2) + (100-300 bytes)	(200 bytes X 3) + (100-300 bytes)
Support for Browser Clients	Yes	Yes	No	No

Table I: Architectural Comparison of Mobile/Cloud Interaction Approaches

message exchange is bounded by the polling interval. Push architectures have artificial rate and size limitations placed on them by the push messaging provider. For example, Google cloud messaging uses a token bucket scheme to rate limit push notifications and enforces a maximum message size of 4K.

B. Analysis 2: Architectural Implementation Complexity

The second analysis focused on the overall complexity of implementing the Model View Controller (MVC) pattern where the model resided in the cloud. We chose the MVC design pattern because it is an extremely common pattern in mobile cyber physical applications. Typically, a model of the physical world, which is updated by sensors on one or more devices, sends events to a controller object, which is responsible for updating a user interface. In many situations, the model represents a physical state that is determined by aggregating sensor data from multiple mobile devices. When a mobile device sends a sensor update and the cloud-based model changes, other dependent mobile device controllers need to be asynchronously notified of the update.

Analysis Design. Apples to apples comparisons of varying code bases are extremely hard. A typical analysis approach would be to report the number of lines of code to implement each of the approaches. SodaCloud handily wins such comparisons. However, we feel that a strict lines of code comparison is inappropriate, since the relative complexity of those lines of code is more important than the total lines of code. Rather than focusing strictly on the number of lines of code we instead analyze the inherent complexities such as user managed threads and message queues for each architecture.

Overview of Architectures. Listings 9-14 present either actual code or pseudo-code to implement the MVC approach with a cloud-based model. For SodaCloud we present the actual code for implementing this pattern since it is concise. For the other architectures, because there is substantial code involved, we instead present pseudo-code for the mandatory steps that custom user-provided code must perform to implement the MVC scenario.

```
Model model = soda.get("theModel", Model.
    class);
model.addListener(new ModelListener(){
  //... listener methods
});
```

Listing 9: Actual SodaCloud Client-side Java Code for Listener Registration & Delivery

```
public class Model {
   private List<ModelListener> listeners_ =
       ...;
   public void addListener(ModelListener l)
       {listeners_.add(l);}
   public void notifyListeners(Change c){
      for(ModelListener l : listeners_){l.
          modelChanged(c);}
   }
}
```

Listing 10: Actual SodaCloud Cloud-side Java Code for Listener Registration & Notification

The HTTP Polling approach, shown in the following two listings, relies on sending HTTP requests to a server in a background thread. The server manages a set of outbound message queues that are polled by clients to get new server-initiated messages. The client-side HTTP Polling thread receives these messages and routes them to the appropriate handler threads to process the incoming messages from the server.

```
[caller thread]
  marshall registration request msg type
[in HTTP polling thread]
  retrieve pending requests from outbound
      queue
  send marshalled HTTP request
  unmarshall new msgs in result
  transfer new msgs to caller thread
      inbound msg queue
[caller thread]
  dequeue msg
  route msg based on type to appropriate
      receiver
  controller is notified of model change
```

	SodaCloud	HTTP Polling	HTTP + Push	HTTP + Push (large msgs)
Total User-managed Threads	0	1 client	1 client + 1 server	1 client + 1 server
Total User-managed Msg Queues	0	1 client + 1-per-client on server	1-client	1-client
Total User-marshalled Msg Types	0	1 client + 1 server	2-client + 2 server	3 client + 3 server
Requires Custom Address Management	No	Yes	Yes	Yes
Object-oriented Abstractions	All Calls	Client-to-Cloud Only	Client-to-Cloud Only	Client-to-Cloud Only

Table II: Cloud-based Model and Device-based View and Controller MVC Implementation Complexity

Listing 11: Condensed Psuedo-code for HTTP Polling Client-side for Listener Registration & Notification Delivery

```
[HTTP request handler thread]
  read HTTP request from client
  unmarshall request msg
  route msg to handler based on type
  check client outbound queue for new msgs
  marshall msgs into HTTP response for
      client
[another thread]
  modify model
  create model change notification
  route notification to client outbound
      queues
```

Listing 12: Condensed Psuedo-code for HTTP Polling Cloud-side for Listener Registration & Notification

The push notification architecture relies on two disjoint communication pathways. The client communicates with the server by marshalling messages into HTTP requests that are sent in a background thread to the server. The server modifies the model in one or more threads and then pushes notifications out to the listening clients using a background thread for sending push notifications. The push sender thread manages converting push messages for the clients into HTTP requests and sending those requests to the push notification server for delivery to the mobile clients. Mobile clients receive incoming push notifications in a distinct thread and then route the incoming messages to other threads to handle any operations or data contained in the messages (*e.g.,* to the display thread to update the UI).

```
   [caller thread]
   marshall registration msg into HTTP
       request
[in HTTP sender thread]
  retrieve pending requests from outbound
      queue
  send HTTP request
[inbound push msg thread (managed by push
    framework)]
  receive push msg
  unmarshall msg
  route msg based on type
  place msg in queue of caller thread
[caller thread]
  unqueue msg
  notify controller of model change
```

Listing 13: Condensed Psuedo-code for HTTP+Push Client-side for Listener Registration & Notification

```
[HTTP request handler thread]
  read request from client
  unmarshall request msg
  route msg to handler based on type
[another thread]
  modify model
  create model change notification
  determine push addresses of listeners
  route msg to outbound push queue
[push sender thread]
  retrieve outbound push msg
  marshall msg into push format
  create HTTP request to deliver push msg
  send HTTP push request to push server
```

Listing 14: Condensed Psuedo-code for HTTP+Push Cloud-side for Listener Registration & Notification

Analysis 2 Results. Table II shows the results of the analysis. The results track key issues of complexity, such as synchronized message queues for exchanging data between threads, user-managed threads, and custom marshalling logic.

V. Related Work

A variety of prior distributed object computing approaches have looked at client/server interactions. Remote procedure calls (RPC), automate and simplify message marshalling and service invocation [9], [10], but do not typically allow exchange of object references and hence cannot easily distribute and implement design patterns, such as MVC. A variety of RPC frameworks are available for mobile devices, such as Google Cloud Endpoints. A variety of publisher/subscriber services [11] have been investigated for mobile / cloud interactions. These services manage message delivery and notification, but are not designed to focus on object/method addressing of messages or support implementation of common software design patterns

A variety of distributed object frameworks have been proposed and implemented, such as Microsoft's distributed component object model(DCOM) [12] that allows communication among software components distributed across a network. Another well-researched distributed object framework is the Common Object Request Broker Architecture(CORBA) [13], which is an open industry standard

defined by Object Management Group(OMG). It enables object-based communication, similar to SodaCloud, but using proprietary protocols [14]. Prior distributed object frameworks have not gained traction in the mobile/cloud interaction space on Android or iOS, primarily we believe because of the lack of support for browser-based clients and reuse of existing HTTP infrastructure. SodaCloud combines key architectural concepts from this prior distributed object framework research with web-socket communication and contextual object discovery to overcome limitations of applying them to mobile/cloud interaction.

Message passing, which push notifications fall under, is another approach to achieve communication between distributed systems. Messages are not passed from client to server directly, instead an intermediate message queue (*e.g.,* push server) is used to store and forward the message. In message passing approach, sender and receiver are decoupled so the sender only need to be responsible for sending the message and does not need to know who consumes the message. Some representatives include MPI [15], SOAP [16]. Recent years, there are techniques proposed specific for cloud to device communication or push notification, such as Google Cloud Messaging or PubNub [17]. As outlined in the paper, the lack of support for intra-app object/method addressing and message delivery adds significant complexity to mobile/cloud interactions.

VI. Conclusion

HTTP is the most common protocol used for mobile/cloud interactions. However, HTTP creates significant challenges for building mobile cyber-physical applications that interact closely with the cloud. Although a variety of distributed object computing approaches and techniques have been developed in past research, they have not gained traction in mobile/cloud applications, due to complexity, lack of support for browser-based clients and legacy HTTP infrastructure, and limited contextual object discovery capabilities. As shown in this paper, many of the past distributed object computing architectural features can be combined with web sockets can contextual object reference discovery to improve the construction of mobile cyber-physical applications that support browser-based clients and legacy HTTP infrastructure.

Acknowledgements

This research was supported by a grant from Siemens Corporate Research.

References

[1] F. B. Luqman and M. L. Griss, "Overseer: A mobile context-aware collaboration and task management system for disaster response," in *The Eighth International Conference on Creating, Connecting and Collaborating through Computing, UC San Diego, La Jolla CA, United States*, 2010.

[2] J. T. B. Fajardo and C. M. Oppus, "A mobile disaster management system using the android technology," *WSEAS Transactions on Communications*, vol. 9, no. 6, pp. 343–353, 2010.

[3] J. White, C. Thompson, H. Turner, B. Dougherty, and D. C. Schmidt, "Wreckwatch: automatic traffic accident detection and notification with smartphones," *Mobile Networks and Applications*, vol. 16, no. 3, pp. 285–303, 2011.

[4] J. Froehlich, T. Dillahunt, P. Klasnja, J. Mankoff, S. Consolvo, B. Harrison, and J. A. Landay, "Ubigreen: investigating a mobile tool for tracking and supporting green transportation habits," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2009, pp. 1043–1052.

[5] H. Bae, M. Golparvar-Fard, and J. White, "Enhanced hd4ar (hybrid 4-dimensional augmented reality) for ubiquitous context-aware aec/fm applications," in *Proceedings of 12th international conference on construction applications of virtual reality (CONVR 2012)*, 2012, pp. 253–26.

[6] M. Lin, N. D. Lane, M. Mohammod, X. Yang, H. Lu, G. Cardone, S. Ali, A. Doryab, E. Berke, A. T. Campbell *et al.*, "Bewell+: multi-dimensional wellbeing monitoring with community-guided user feedback and energy optimization," in *Proceedings of the conference on Wireless Health*. ACM, 2012, p. 10.

[7] A. Thiagarajan, L. Ravindranath, K. LaCurts, S. Madden, H. Balakrishnan, S. Toledo, and J. Eriksson, "Vtrack: accurate, energy-aware road traffic delay estimation using mobile phones," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2009, pp. 85–98.

[8] I. Fette and A. Melnikov, "Rfc 6455: The websocket protocol," RFC, IETF, December 2011. See http://www.ietf.org/rfc/rfc6455.txt, Tech. Rep., 2011.

[9] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 1, pp. 39–59, 1984.

[10] A. Wollrath, R. Riggs, and J. Waldo, "A distributed object model for the javaˆ tˆ m system," *Computing Systems*, vol. 9, pp. 265–290, 1996.

[11] M. Volter, M. Kircher, and U. Zdun, *Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware*. Wiley. com, 2005.

[12] N. Brown and C. Kindel, "Distributed component object model protocol–dcom/1.0," *Online, November*, 1998.

[13] O. M. Group, *The Common Object Request Broker (CORBA): Architecture and Specification*. Object Management Group, 1995.

[14] D. C. Schmidt, A. S. Gokhale, T. H. Harrison, and G. Parulkar, "A high-performance end system architecture for real-time corba," *Communications Magazine, IEEE*, vol. 35, no. 2, pp. 72–77, 1997.

[15] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel computing*, vol. 22, no. 6, pp. 789–828, 1996.

[16] J. Snell, D. Tidwell, and P. Kulchenko, *Programming Web services with SOAP*. O'reilly, 2009.

[17] "http://www.pubnub.com/."