PERFORMANCE EVALUATION OF TRANSIT DATA FORMATS ON A MOBILE DEVICE

Sean J. Barbeau (Corresponding Author)
Center for Urban Transportation Research
University of South Florida
Tampa, FL 33620
813-974-7208
barbeau@cutr.usf.edu

Tatu Saloranta
FasterXML, LLC
Seattle, WA 98103
tatu@fasterxml.com

November 15, 2013

5,189 words + 9 figures (2,250) = 7,439 words

Abstract

Over the last decade mobile devices have changed how we access transportation information. However, many past data formats used in information services have been designed without the processing and energy constraints of mobile devices in mind. Newer standards, such as the Service Interface for Real-time Information (SIRI) v2.0, are including mobile-friendly features that reduce the processing overhead on mobile devices. Coupled with the advances in mobile device processing capabilities, many believe that performance is no longer an issue of concern in modern mobile phones. This paper presents an evaluation of the SIRI data format on a mobile device, which indicates that not only is performance still an issue, but that app developers should carefully consider certain software design choices to avoid exposing mobile users to extensive wait times (e.g., wait for real-time transit arrival information). The results also demonstrate that information service providers should always offer mobile-friendly interfaces (i.e., RESTful web services with JSON encoding) when possible. The benchmarking software is made available as an open-source application so that others can perform their own experiments, and so that app developers can use this library as a foundation for building new applications based on the SIRI format.

1. INTRODUCTION

Mobile devices have become ubiquitous over the last decade. In late 2011, there were an estimated 5.9 billion cellular subscriptions worldwide with a global penetration rate of 87% [1]. In the United States, as of December 2012 there were 326.4 million cellular subscriptions, a penetration rate of 102.2% (indicating that some users have multiple subscriptions) [2]. As a result, the population's reliance on such devices for completing everyday tasks has increased. According to a 2009 survey, 82% of Americans never leave their house without their phone, and 42% stated "they cannot live without their phone" [3].

Since mobile devices aren't restricted to a single location, they are natural tools that travelers can use to access real-time information during a commute. Research has shown that transit riders with access to real-time information wait around 2 minutes less for a bus than those without real-time information [4]. In the same study, riders with real-time information perceived their wait time to be around 30% shorter than riders without real-time information, indicating that real and perceived benefits both exist [4].

As is common in the early years of technology growth, mobile device and mobile app evolution has been organic, with a variety of companies and individuals filling the need for various transportation services as they emerge, often using a variety of proprietary data formats. Eventually, however, the issue of interoperability between systems must be addressed. For example, many of today's mobile transit apps developed in one city cannot easily function in another city due to differing real-time data formats. Standards, or formats agreed upon by a group of stakeholders, can potentially benefit the industry by establishing a consensus on communication protocols. This would, for example, enable one city to establish a real-time data feed that is identical to another city, so that mobile apps for first city would be transferrable to the second.

When examining data formats in the context of mobile device communication, key differences from desktop computers have been observed in the areas of processing power, communications, and energy constraints [5, 6]. Due to smaller form-factors, mobile devices typically have lower-powered processors that generate less heat, resulting in lower processing performance when compared to a desktop computer. Additionally, mobile devices have a constrained wireless communication channel, both in terms of speed, which is slower than a typical wired broadband connection, and in terms of the amount of allowed data transfer, which is typically paid for by the wireless subscriber based on a tiered data plan (i.e., transferring more data costs more money). Finally, mobile devices have significant energy limitations – they are battery-powered, and all application actions (e.g., program execution, wireless transmissions) consume energy.

New data standards should recognize the importance, and limitations, of mobile devices. As of August 2013, the European Committee for Standardization (CEN) is in the process of ratifying the new v2.0 version of the Service Interface for Real Time Information (SIRI) (CEN/TS 15531) standard [7]. SIRI v2.0 includes a new "mobile-friendly" version of this standard, or "SIRI Lite", based on the experience of the Metropolitan Transit Authority (MTA) when implementing the MTA Bus Time real-time transit information system in New York [8].

With new "mobile friendly" data formats becoming available, and with the processing capabilities of new smart phones on the market approaching 2.0 GHz and eight-core CPUs, one may think that the burden on app developers for implementing smart and efficient apps is lifted. However, this is far from true.

This paper presents a performance evaluation of device-to-server communication using the SIRI data format on a modern smart phone. The results indicate that not only is performance still an issue, but that app developers should carefully consider certain software design choices to avoid exposing mobile users to extensive wait times (e.g., for real-time transit arrival information). The results also demonstrate that information service providers should always offer mobile-friendly interfaces when possible. Optimizations to reduce user wait times are also presented.

Fast processing of data from a server is important because it has a direct effect on the amount of time a user has to wait before they can see results fetched from the internet. For example, when a user removes a phone from their pocket, opens the device, and requests updated transit arrival information for their current location, the device must retrieve this information from a server (e.g., a SIRI API). The user will be shown a "Please wait…" screen until the device receives and completely parses a server response. Slower parsing also typically results in longer CPU, screen, and radio usage, which all negatively affect battery life.

The benchmarking software used for these experiments is made available as an open-source application so that others can perform their own experiments, and so that app developers can use this library as a foundation for building new applications using the SIRI format [9].

2. DATA COMMUNICATIONS FOR MOBILE DEVICES

Fast processing of data from a server is important because it has a direct effect on the amount of time a user has to wait before they can see results fetched from the internet. For example, when a user removes a phone from their pocket, opens the device, and requests updated transit arrival information for their current location, the device must retrieve this information from a server (e.g., a SIRI API). The user will be shown a "Please wait…" screen until the device receives and completely parses a server response. Slower parsing also typically results in longer CPU, screen, and radio usage, which all negatively affect battery life.

When discussing device-to-server communication and performance, there are two typical high-level design choices for a web service, or API:

1. The protocol used to transfer data
2. The format of the data being transferred

Protocols

The Hypertext Transfer Protocol (HTTP) underlies most internet communication, and therefore typically plays a role in most API designs [10]. However, a second protocol, SOAP, emerged in the early 2000s to support advanced enterprise server-to-server communication [11]. SOAP uses the Extensible Markup Language (XML) to define protocol properties.

While SOAP has proven useful for advanced server-to-server communication, it is not well suited for mobile devices. The extensive use of XML results in a large overhead being added to each message.

An alternate, simpler protocol for web services, termed "RESTful" web services, is to rely only on HTTP in a state-less design, which does not require advanced record-keeping on the mobile device or server during communication and does not use XML in the protocol itself. Previous research has shown that RESTful web services have a significantly less impact on mobile devices resources [5, 6] than SOAP-based web services. One of the improvements to the SIRI v2.0 "SIRI Lite" format was the inclusion RESTful query support. Since the benefits of

RESTful web services over SOAP-based web services has been demonstrated in previous research, this paper focuses primarily on data formats.

Data Formats

After a protocol that defines the order and properties of data exchange is chosen, the actual format of the data must be defined. Two popular data formats for web services are XML and Javascript Object Notation (JSON) [12].

The following is a JSON-formatted response from the MTA Bus Time RESTful SIRI API:

```
{Siri: {
  ServiceDelivery: {
    ResponseTimestamp: "2012-08-21T12:06:21.485-04:00",
    VehicleMonitoringDelivery: [
      {
        VehicleActivity: [
          {
            MonitoredVehicleJourney: {
              LineRef: "MTA NYCT_S40",
              DirectionRef: "0",
              FramedVehicleJourneyRef: {
                DataFrameRef: "2012-08-21",
                DatedVehicleJourneyRef: "MTA NYCT_20120701CC_072000_S40_0031_S4090_302"
              },
              JourneyPatternRef: "MTA NYCT_S400031",
              PublishedLineName: "S40",
              OperatorRef: "MTA NYCT",
              OriginRef: "MTA NYCT_200001"
            } } ] } ] } }
```

The following is a XML-formatted response from the MTA Bus Time RESTful SIRI API:

```
<Siri xmlns:ns2="http://www.ifopt.org.uk/acsb" xmlns:ns4=http://datex2.eu/schema/1_0/1_0 xmlns:ns3="http://www.ifopt.org.uk/ifopt"
xmlns="http://www.siri.org.uk/siri">
  <ServiceDelivery>
    <ResponseTimestamp>2012-09-12T09:28:17.213-04:00</ResponseTimestamp>
    <VehicleMonitoringDelivery>
      <VehicleActivity>
        <MonitoredVehicleJourney>
          <LineRef>MTA NYCT_S40</LineRef>
          <DirectionRef>0</DirectionRef>
          <FramedVehicleJourneyRef>
            <DataFrameRef>2012-09-12</DataFrameRef>
              <DatedVehicleJourneyRef>MTA NYCT_20120902EE_054000_S40_0031_MISC_437</DatedVehicleJourneyRef>
          </FramedVehicleJourneyRef>
          <JourneyPatternRef>MTA NYCT_S400031</JourneyPatternRef>
          <PublishedLineName>S40</PublishedLineName>
```

```
                <OperatorRef>MTA NYCT</OperatorRef>
                <OriginRef>MTA NYCT_200001</OriginRef>
              </MonitoredVehicleJourney>
            </VehicleActivity>
          </VehicleMonitoringDelivery>
        <ServiceDelivery>
      </Siri>
```

One of the improvements to the SIRI v2.0 “SIRI Lite” format was the inclusion of non-XML encoding, including JSON.

3. METHODOLOGY

The Jackson JSON and XML processor [13] was chosen for these experiments because previous tests have shown that it outperforms other parsers [14, 15], it supports both JSON and XML in a single parser, and it is open-source so that the internal functionality can be examined. It should be noted that certain Java libraries had to be modified so that Jackson would function properly on Android, due to differences in Java for Android versus desktop computers [16]. While the results discussed in this paper are specific to tests performed using an Android application on an Android device, the same general constraints and design considerations should also apply to other mobile platforms such as Apple iOS and Windows Phone.

The SiriRestClientUI app [17] was used to perform benchmarks of JSON vs. XML parsing using the SiriRestClient library [9] on a Samsung Galaxy S3 SPH-L710 with Android 4.1.1, 1.5 GHz dual core processor, 2GB RAM (Power saving mode off). Jackson 2.1.2 with Aalto 0.9.8 was used. The Jackson Internal HTTP connection was used, as well as the ObjectReader for JSON parsing. These tests were performed on the University of South Florida (USF) WiFi network. Results from SpeedTest.NET Android app at time of test were 51,353 kbps down, 49,554 kbps up, and ping of 16ms.

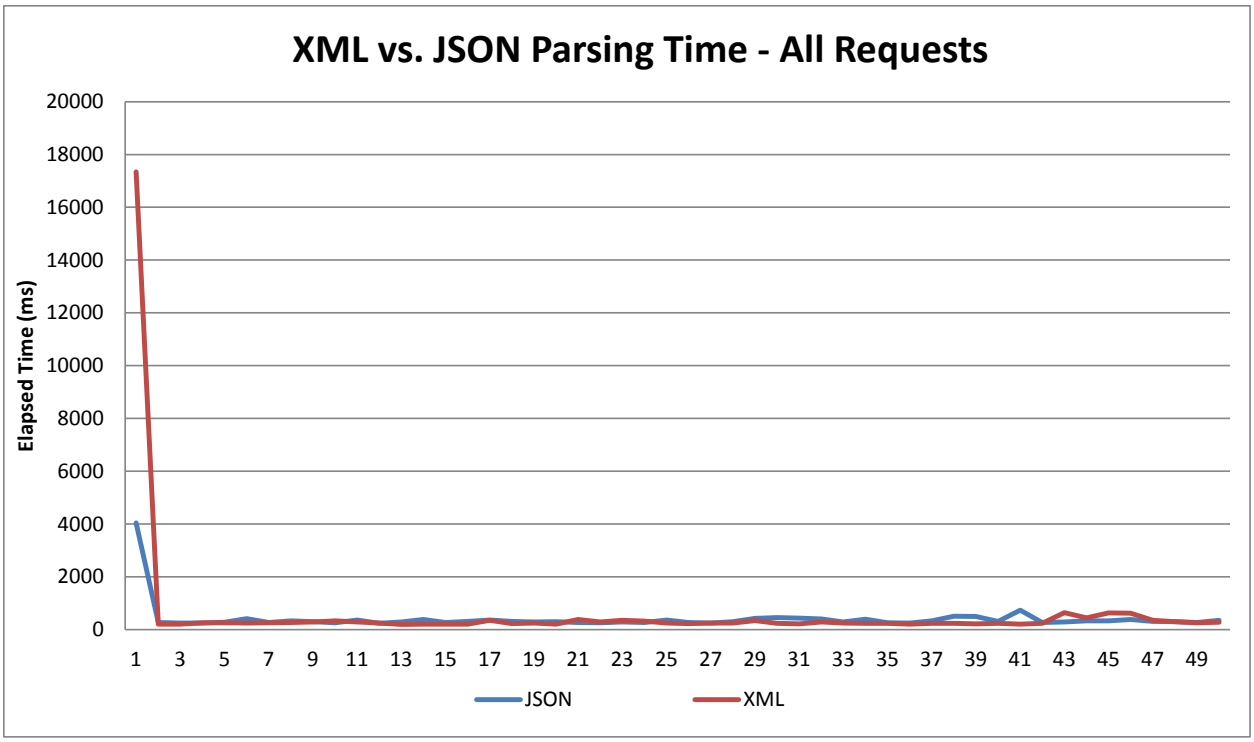

FIGURE 1 - XML vs. JSON Parsing - Elapsed Time from Request to Parsed Response for 50 Requests

50 requests were performed back-to-back using the MTA BusTime SIRI StopMonitoring API. A timestamp recording feature in the SiriRestClientUI app and SiriRestClient library captured how long a request took, from when the request was issued to when a Siri object became available from Jackson.

4. RESULTS

The elapsed time from request to parsed response for 50 sequential requests is shown in FIGURE 1. There is a substantial difference between the "cold start" times (i.e., the first request) for both JSON and XML. The time for the XML cold start response is almost 18 seconds, over 4 times as long as the JSON cold start response (approximately 4 seconds). After the cold start, the differences between the response times for JSON and XML are much smaller.

FIGURE 2 shows the summary statistics for XML and JSON parsing time for this test. JSON outperforms XML in average response time of 401ms, vs. the XML average response time of 625ms. 95th percentile of elapsed times is closer, with JSON having a 95th percentile of 501ms and XML having a 95th percentile of 626ms. The increase in standard deviation for XML response time reflects the initial large cold start value that is substantially larger than the following warm starts. The size of the JSON response was approximately 4KB, with the XML response being approximately 5KB.

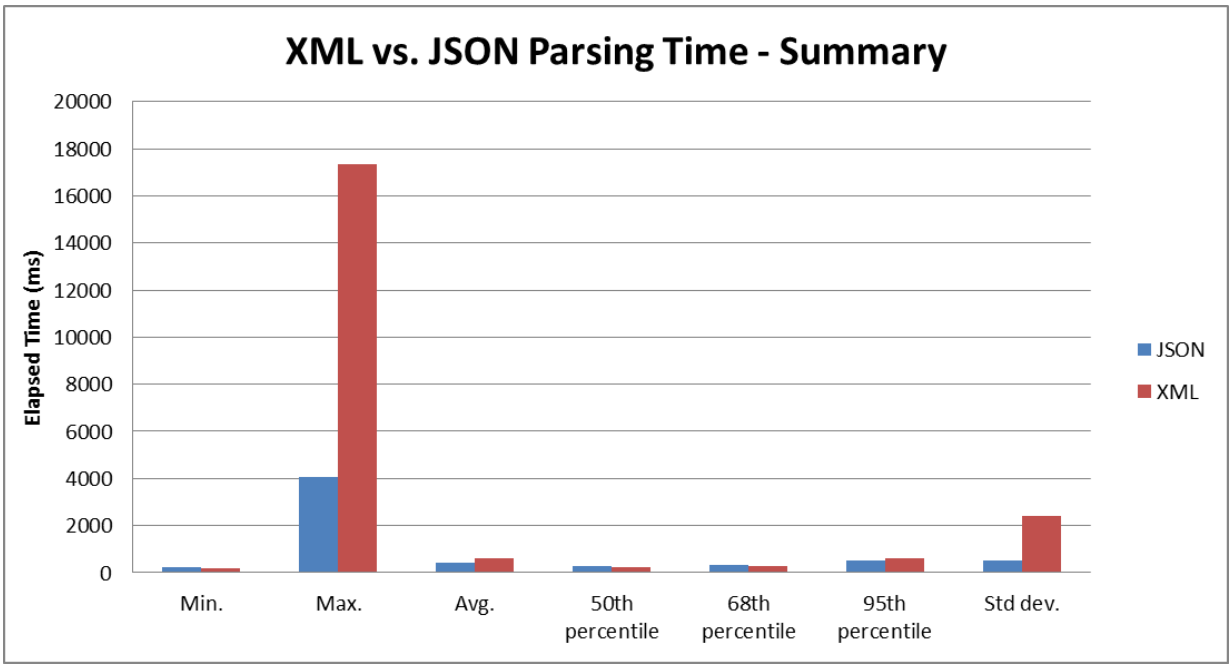

FIGURE 2 - XML vs. JSON Parsing - Summary Statistics for the Elapsed Time of 50 Requests

5. DISCUSSION

The first execution of a request to the server (i.e., a cold start) typically takes much longer than subsequent requests (i.e., warm starts). This is because Jackson will dynamically construct Class model from Java class definitions the first time deserializers are needed (typically when the application initiates the request and the readValue() method is called). Here, JSON yields significant better performance than XML - JSON performance is over 4 times faster than XML with a time difference of 14 seconds. This is likely due to the additional annotation accesses required for XML, an area in which Android currently performs poorly [18]. However, poor XML performance when compared to JSON is not unique to Android. Other performance evaluations for desktop Java virtual machines have found similar relationships, with JSON processing outperforming XML processing for the same encoded content [19]. This likely indicates that XML processing for cold starts will typically perform worse than JSON processing on any Java virtual machine, and possibly on virtual machines for other programming languages as well.

After this initial cold start, Jackson will typically parse subsequent responses more quickly for both JSON and XML. As stated above, JSON has a slight performance advantage for warm starts too - an average of 224ms faster than XML. Since recent human-computer interaction studies have indicated that users can perceive time differences of 100ms when waiting for a response [20], JSON still yields a noticeable performance increase for warm starts from the user's perspective.

On Android, the warm start state can persist even if the app is closed by the user and re-opened. Android devices will typically keep recently closed apps in memory as a cached background process to enhance future startup performance. When the user "starts" the app, it is actually loading the application from the cached process in memory, which loads all the Jackson data structures needed to perform quick parsing on warm starts without needing to re-initialize them from a cold start state. As a result, when comparing tests, it’s important to note that the first

execution of the app will typically show significantly worse performance than subsequent warm starts.

While the warm start state provides a significant advantage in response time performance, it unfortunately cannot be relied upon for consistent performance increases after an app is started on the device. Android may remove a cached process from memory if the platform is running low on memory. Given the multitasking that typically occurs on most cell phones, especially in a scenario where the user is waiting for a bus to arrive, performing tasks such as checking email, internet browsing, or using social networking apps may result in the real-time transit app being removed from the app cache. At this point, the app will revert to the cold start state and there may be a significant delay in retrieving transit data. The size of the SiriRestClientUI cached process was observed to typically be between 24-31MB, which is large for a frequently used app. For comparison, on a Samsung Galaxy S3 the following apps had the following cached sizes - Calendar app = 7.2MB, Clock = 16MB, Google+ = 17MB, Maps = 13MB. Since the largest non-system processes are typically the first targets for process cache eviction, it is likely that this app would frequently be restored to a cold start state.

The above observations lead to the development of a manual caching strategy for Jackson objects on Android in an attempt to consistently reduce the cold start penalty for response times. The follow section discusses the results of these Jackson object manual caching optimizations.

6. OPTIMIZATIONS USING PARSER OBJECT CACHING

To improve cold start performance of retrieving and parsing the SIRI responses using Jackson, further control over the caching process must be achieved. A strategy called "Pseudo-warm starts" was developed, which is defined as follows:

- Pseudo-warm start = An artificial warm-start by the app, where the Jackson object is manually cached to persistent memory by the app and read from memory during app startup (in what would otherwise be a cold start).

FIGURE 3 shows the stages of application execution for A) cold starts, B) warm starts, and C) pseudo-warm starts.

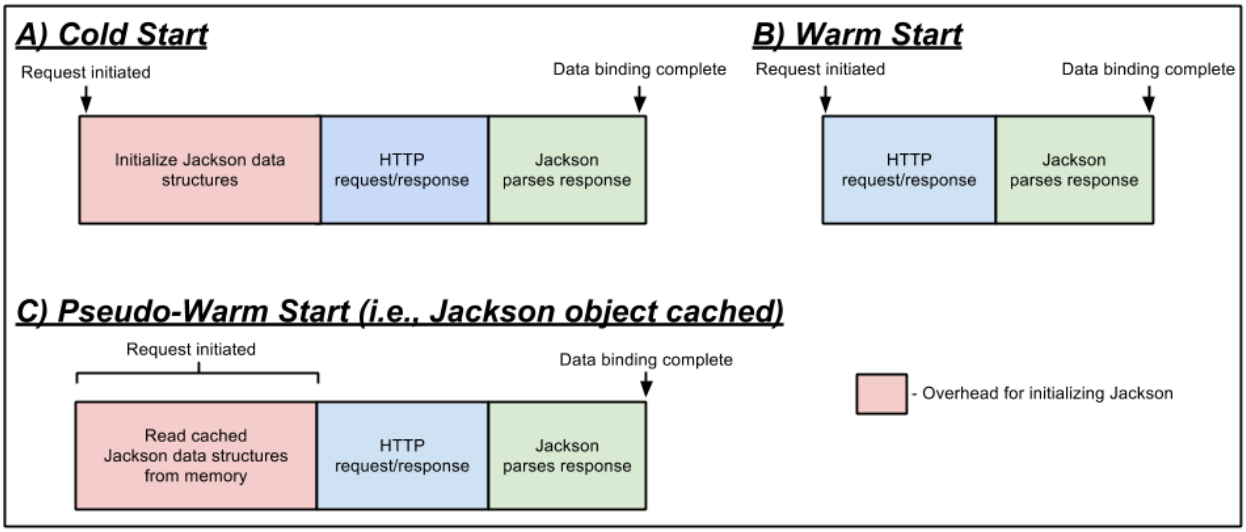

FIGURE 3 - The Stages of App Execution for Cold, Warm, and Pseudo-warm Starts

The red-shaded blocks (i.e., the first block in FIGURE 3A and FIGURE 3C) indicate the overhead required to initialize Jackson (note that warm starts, FIGURE 3B, do not have this overhead). In FIGURE 3C, the normal Jackson initialization process is replaced with reading the cached Jackson objects that were initialized in a previous application execution and then written to a persistent cache (i.e., the pseudo-warm start). A note-worthy property of pseudo-warm starts is that the cache read can be initiated on application start-up without requiring user initiation. This flexibility is indicated in FIGURE 3 via the "Request initiated" bracket over the entire cache read period, as the request may be initiated by the user at any point during the cache read. The flexibility of when the cache read is initialized is important in later discussions of this optimization.

The goal of the next set of experiments was to evaluate whether the cache read used in pseudo-warm starts was faster than the normal Jackson initialization process in cold starts.

Methodology

The SiriRestClientUI app with the SiriRestClient library was again used to perform benchmarks of JSON vs. XML parsing on the same Samsung Galaxy S3 device and with the same settings as the previous tests. Results from SpeedTest.NET Android app at time of test were 52,176 kbps down, 78,721 kbps up, and ping of 15 ms.

30 requests were performed using the MTA BusTime SIRI StopMonitoring API. The following steps were repeatedly taken to reset to a cold start state between each cold start test, and to the pseudo-warm start state for the pseudo warm start tests:

A. A cold start response time was measured,
B. Caching was turned on in the app and another request was made so the app caches the Jackson object used to make requests to persistent memory - cache write time and cached Jackson object size were measured,
C. The Android cached process was manually removed from memory via the Application Manager to reset to a cold start state,
D. A pseudo-warm start response time was measured (= Time to read cached object + elapsed response time).
E. Caching was turned off in the app, and the Android cached process was manually removed from memory via the Application Manager to reset to a cold start state

Results

First, the performance of cold vs. pseudo-warm starts when using XML to format the server response was examined. FIGURE 4 shows the elapsed time of 30 cold and pseudo-warm start tests, and FIGURE 5 shows the summary of the results from these tests.

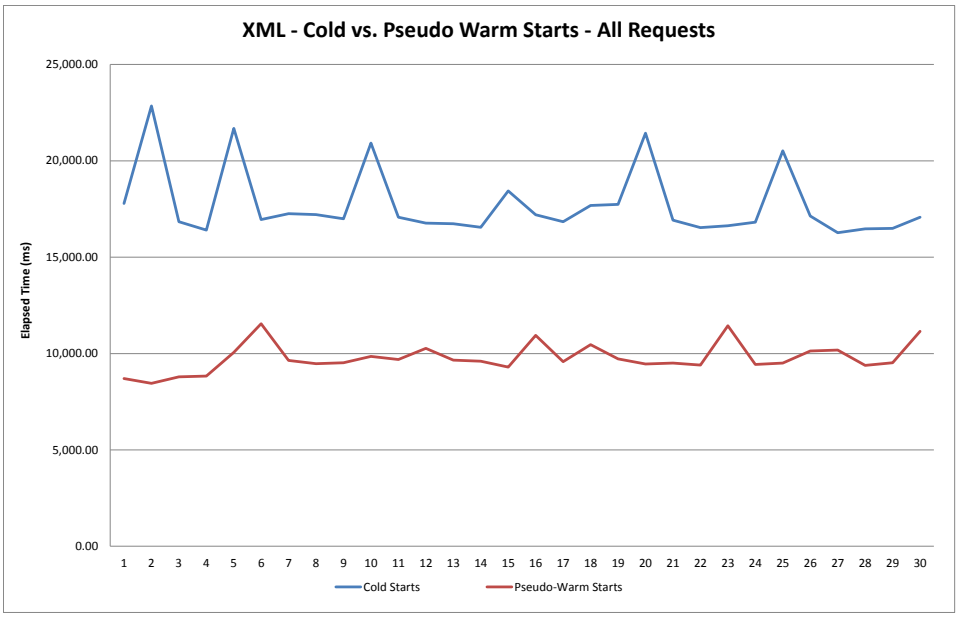

FIGURE 4 - Cold vs. Pseudo-Warm Start performance for XML responses from 30 requests

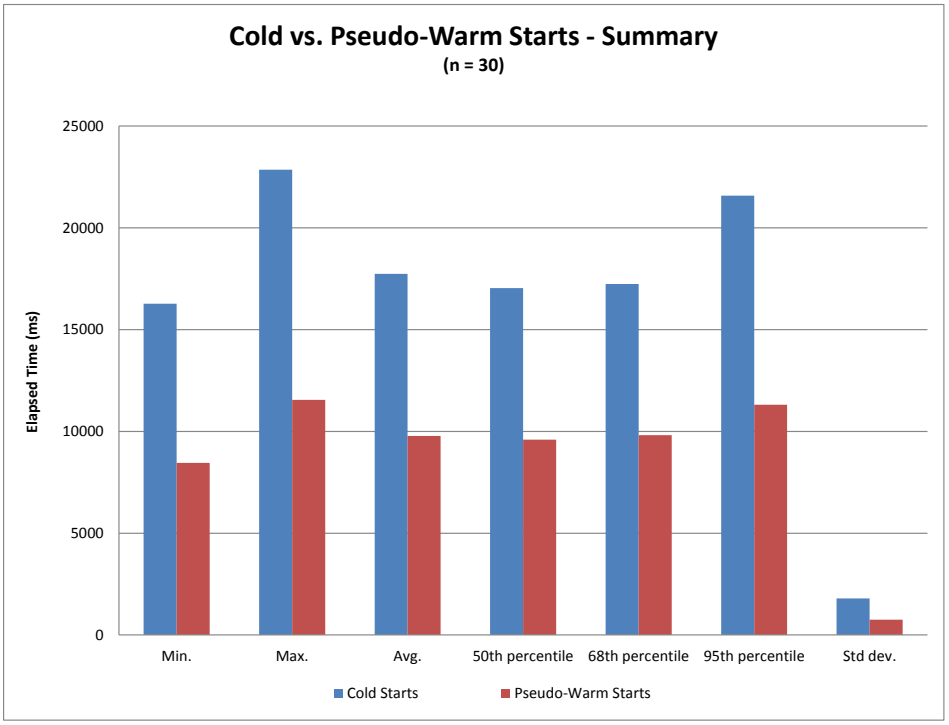

FIGURE 5 - Cold vs. Pseudo-Warm Start performance for XML responses - Summary

Because of the large cold start penalties, improve the performance of XML parsing via pseudo-warm starts and caching is substantially improved. Cold starts had an average elapsed time of 17.7 seconds, and pseudo-warm starts had an average of only 9.7 seconds, an 8 second improvement. Overall, pseudo-warm starts thereby yield an average of a 44% performance increase over cold starts, a dramatic improvement.

Next, the performance of cold vs. pseudo-warm starts when using JSON to format the server response was examined. FIGURE 6 shows the elapsed time of 30 cold and pseudo-warm start tests, and FIGURE 7 shows the summary of the results from these tests.

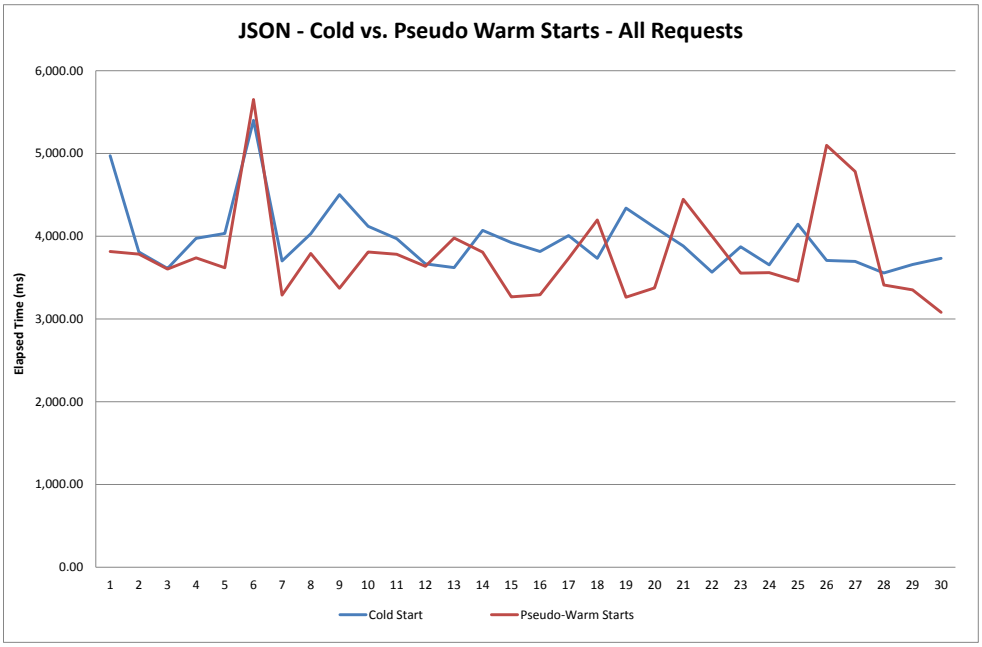

FIGURE 6 - Cold vs. Pseudo-Warm Start performance for JSON responses from 30 requests

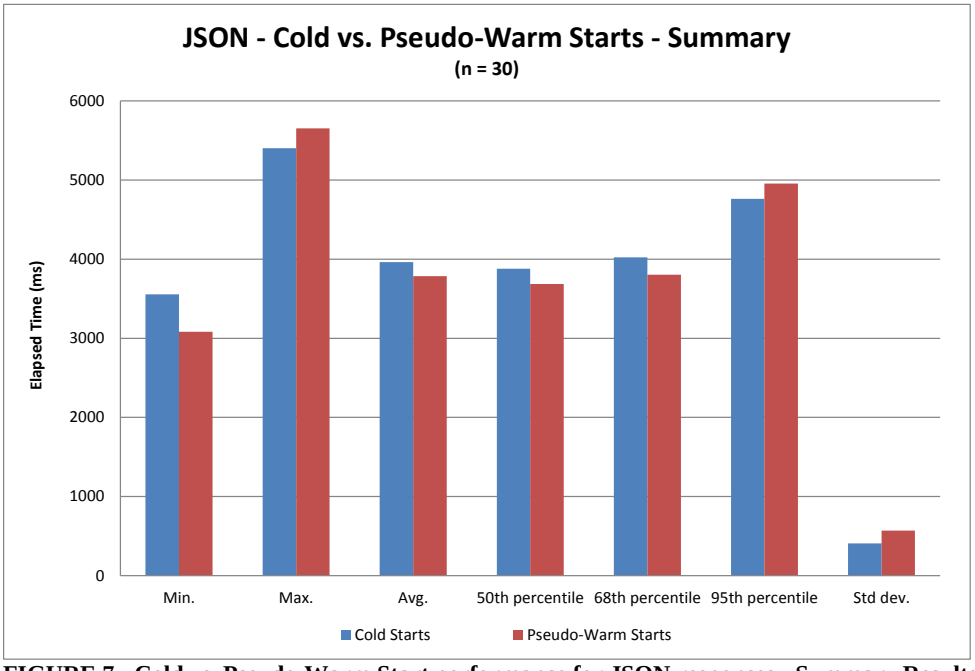

FIGURE 7 - Cold vs. Pseudo-Warm Start performance for JSON responses - Summary Results

Pseudo-warm starts perform slightly better than cold starts, with an average elapsed time of 3,785ms vs. the cold start average of 3,963ms - a difference of 178ms on average. This difference may still be noticeable to the user [20], but is not nearly as dramatic as the improvement of the XML pseudo-warm start.

Overall, for JSON the pseudo-warm start amounted to an average 3.96% performance increase over the cold start.

Discussion

The dramatic improvement in XML parsing performance using pseudo-warm starts indicates that the time required to read a cached Jackson object from mobile device persistent memory is far less than the time required to newly instantiate that same object. Therefore, a huge 44% performance increase on average can be produced by using pseudo-warm starts.

JSON pseudo-warm start performance improvements (3.96% on average), however, are not nearly as impressive as their XML counterparts. This difference is partially due to the fact that XML cold starts are significantly larger than JSON cold starts to begin with (by a factor of 4), which gives the XML pseudo-warm start a much greater margin of potential improvement.

After reviewing these results, one may think that a pseudo-warm start implementation for JSON parsing isn't worth the effort. However, a significant advantage of pseudo-warm starts over cold starts hasn't yet been discussed - the potential to hide part of the delay from the user by beginning the cache read when the app is first started.

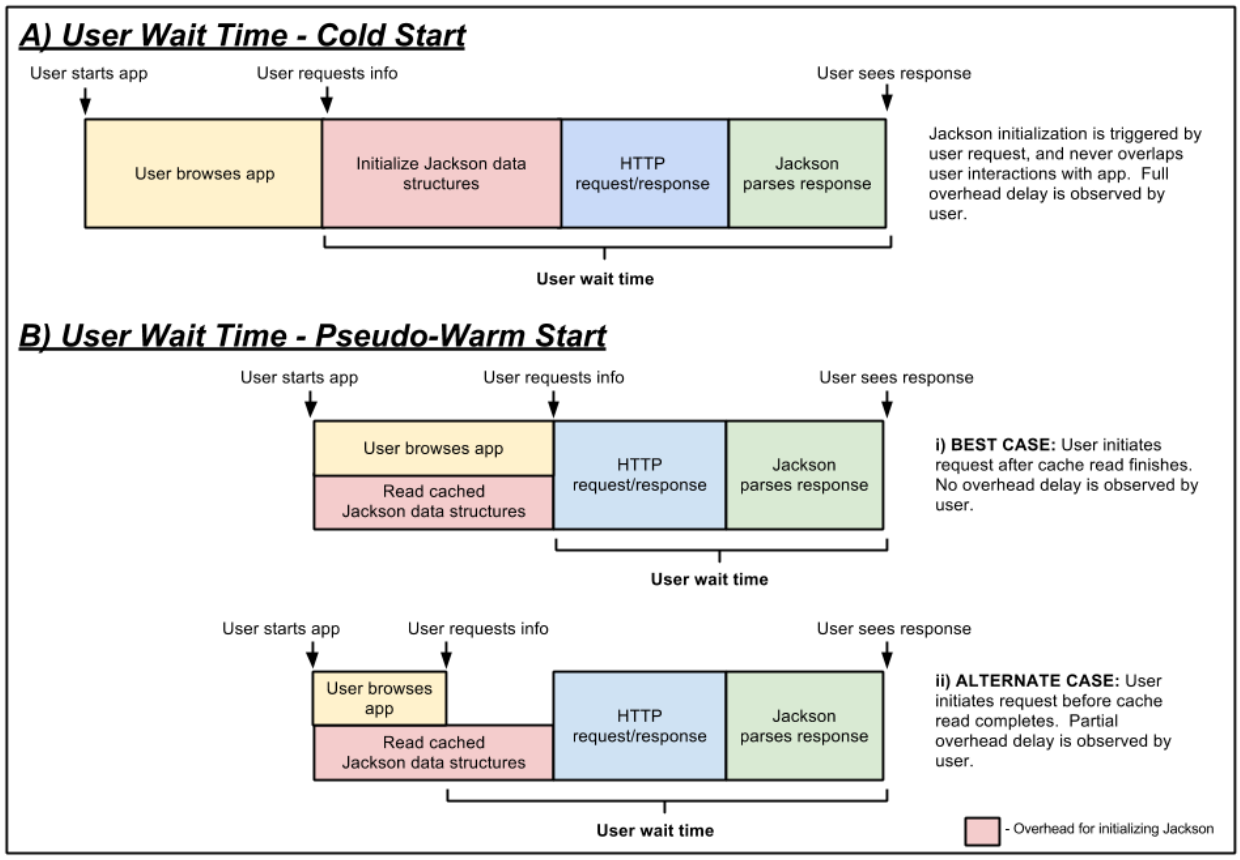

FIGURE 8 - With A) cold starts, the user always observes the fully overhead delay, but with B) pseudo-warm starts the overhead delay can be hidden from the user

FIGURE 8 shows how user interactions (shaded in yellow) occur in context of cold starts (FIGURE 8A) compared with pseudo-warm starts (FIGURE 8B).

As mentioned earlier, the cache read used in pseudo-warm starts can be initiated immediately upon application start up and can execute in the background while the user is browsing through the application. Therefore, using pseudo-warm starts, a considerable amount of the overhead time may pass before the user actually triggers a request to the server (e.g., for real-time transit arrival information). In the best-case scenario, shown in FIGURE 8Bi, the entire cache read finishes prior to the user initiating a request, and the user does not observe any delay. In this case, pseudo-warm starts are equivalent to warm starts in performance. Cold starts, on the other hand, cannot hide any of this overhead wait time from the user - Jackson initialization must always start when the user initiates a server request, as shown in FIGURE 8A.

An alternate pseudo-warm start scenario is shown in FIGURE 8Bii. Here, the user browsers the app while the cache read starts, but initiates the request to the server before the cache read can complete. In this situation, the user would observe a partial delay, depending on how much time has elapsed and how long the cache read will take to complete.

When considering that the cache read time can now be partially or completely hidden from the user, the seemingly small improvements in JSON pseudo-warm start performance suddenly become quite large. FIGURE 9 shows the differences between pseudo-warm starts and cold starts if the user spends the entire cache read time browsing through the app.

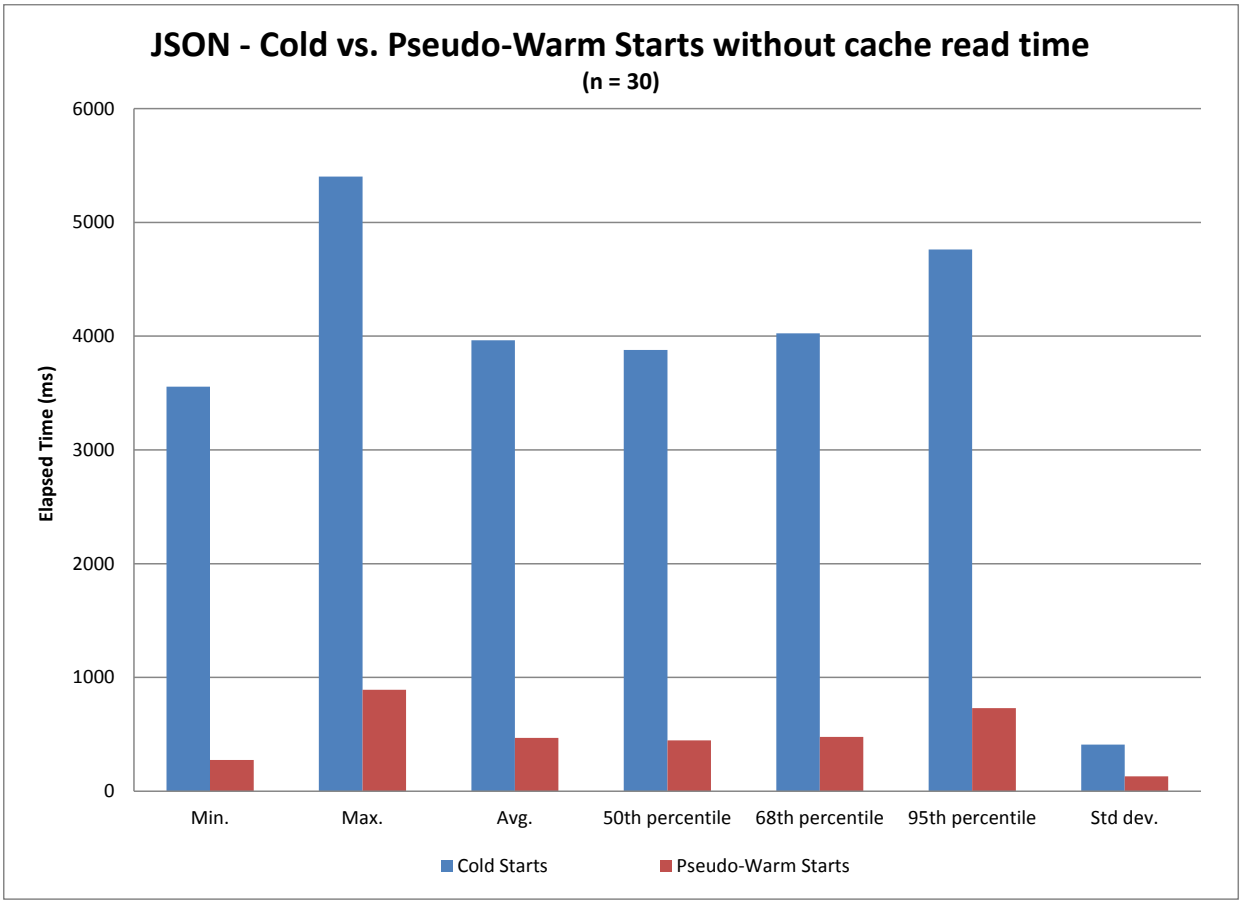

FIGURE 9 - If user interactions with the app hide the cache read latency, JSON pseudo-warm starts perform significantly better than cold starts

Assuming that this cache read time is hidden from the user, the average JSON pseudo-warm start elapsed time is 469ms, compared to nearly 4 seconds of cold start time. The average cache read time in this test was 3.3 seconds, meaning the user would need to spend 3.3 seconds in the app before requesting real-time transit information for the cache read to remain completely hidden. Given the initial app start-up process and other activity of the user before they may retrieve real-time transit information, the expectation that at least some of the cache read will be hidden seems reasonable.

Revisiting XML pseudo-warm starts while considering the potential to hide the cache read yields even greater performance increases over cold starts. Average pseudo-warm starts improve to 473ms, versus average cold starts of 17.7 seconds. However, one must also consider that to fully hide the cache read for XML parsing, the user would need to spend an average of 9.3 seconds in the app before triggering a request to the server. This is a much longer amount of time than that required for JSON, and therefore less of the XML cache read time will be hidden from the user on average. The longer cache read time for XML when compared to JSON may be explained by the larger cache file size for the XML (1,200kB) vs. JSON (260kB).

One drawback to the caching strategy of pseudo-warm starts over cold starts is the persistent memory consumed by the cache file. However, the observed sizes of the cache files should be negligible for most smart phones.

7. RELATED WORK

The strategy of pseudo-warm starts is just one potential avenue for improving user experience in mobile applications that use real-time information from a server. This paper focuses on reducing the amount of time necessary to retrieve new real-time information from a server when an update needs to be retrieved. Response caches, available on Android 4.0 and up, can be used to avoid retrieving a full response to the mobile device when no change in the data has occurred since the last request. Compression can also be used to reduce the size of HTTP data transfers (although the computational costs of device-side decompression must also be taken into account). The refresh interval on the device can also be adjusted to avoid querying the server too frequently. As previously mentioned, the protocol used to transfer information also affects performance. Past research has shown that RESTful web services are heavily preferred to SOAP-based web services on mobile devices [5, 6].

8. FUTURE WORK

Another strategy for hiding the latency for cold starts from the user is to initiate a "dummy" read of a small bit of data on startup, which will force initialization of many of the internal Jackson data structures used for deserialization. While the dummy read isn't expected to increase the general initialization performance, it does offer the advantage of being able to hide some of the latency from the user vs. a normal cold start. Future work could compare the performance of dummy reads against cache read times to quantify the differences.

The number of cores in mobile processors continues to increase, with eight core processors just around the corner for mobile phones. Further experimentation could also examine the potential for parallelizing computations on Android to speed up both JSON and XML processing. However, such speedups are not expected to effect the relative relationship between the time required for JSON and XML processing when the same general processing model is used for both (which is the case in this paper).

Additional future benchmarking could also be performed in a multi-tasking environment where multiple applications are being context-switched as they are brought to the foreground and

returned to the background during JSON and XML process. However, the authors do not expect the general performance results presented in this paper to differ drastically in this situation. The parsing and generation of data formats like JSON and XML is a relatively simple linear CPU-intensive (and CPU-bound) process, and as such concurrency issues are rarely problematic when benchmarking this type of performance. As a result, linear single-threaded tests are usually sufficient when performing such benchmarking. Additionally, in many real-world time-sensitive application scenarios (e.g., a user checking the estimated arrival time while waiting at a bus stop), the user is waiting for the result to be processed while the application is in the foreground, in a very similar design to the tests in this paper.

Different size responses could also be analyzed in future work, to determine if this would affect the relative relationship between JSON and XML processing time (e.g., if JSON is better for smaller datasets, while XML is better for larger datasets). The authors of this paper believe that the general processing relationships of JSON being faster than XML would continue to hold, no matter the dataset size, as the processing models for JSON and XML are very similar at conceptual and implementation level.

Finally, implementation-specific future work could also be examined. Tests presented in this paper used Jackson v2.1.2. Newer versions of Jackson are now available and should be used in future benchmarking tests. Further investigation into the time required to access annotations on Android revealed an issue on the Android platform [18]. It appears that this issue is now fixed in the Android Open-Source Project, but has not yet been included in any Android releases (4.3 and lower) for Android devices. Future benchmarking should be performed on releases of Android with this improvement to see if performance increases – however, as noted earlier, other experiments on desktop computer using Java have yielded similar performance benefits of JSON over XML, and therefore improvements in Android are not expected to drastically affect the overall results of the experiments presented in this paper.

9. CONCLUSIONS

When using real-time information services, users must often wait for their phone to retrieve the latest real-time information from a server. This paper presented an evaluation of the effect of data formats on the time required for a mobile device to retrieve updated information from a server, in the context of real-time estimated arrival information for public transportation. The results indicate that app developers should carefully consider certain software design choices to avoid exposing mobile users to extensive wait times (e.g., for real-time transit arrival information). The results also demonstrate that information service providers should always offer mobile-friendly data (i.e., RESTful web services with JSON encoding) when possible.

JSON was shown to be a preferred data transfer format over XML for mobile devices. Average performance for cold starts (i.e., when the user first starts the mobile app) was over 4 times faster for JSON than XML, with an average time difference of 14 seconds. JSON also had a noticeable performance advantage in warm starts, being an average of 224ms faster than XML.

An optimization strategy, pseudo-warm starts, was also presented that aims at reducing the large performance penalty of cold starts. Pseudo-warm starts, which use cached Jackson objects instead of re-initializing the objects on app startup, provide a dramatic increase in performance for XML parsing, reducing elapsed parsing time from an average of 17.7 seconds to an average of only 9.7 seconds, an 8 second improvement. Pseudo-warm start improvements for JSON were more modest, with an average elapsed time of 3,785ms vs. the cold start average of 3,963ms - a difference of 178ms on average.

Finally, the potential impact of pseudo-warm starts vs. cold starts in the context of user-observable performance was discussed. Assuming that the entire cache read time is hidden from the user, the average JSON pseudo-warm parsing start elapsed time improves to 469ms, compared to nearly 4 seconds of cold start time. Similarly, assuming a fully hidden cache read, average XML pseudo-warm starts further improve to 473ms, versus average cold starts of 17.7 seconds. However, given that XML cache reads take an average of 9.3 seconds, versus average JSON cache read of 3.3 seconds, it may not be realistic to completely hide XML cache reads from the user.

10. ACKNOWLEDGMENTS

The research described in this paper was funded by the National Center for Transit Research at the University of South Florida.

11. REFERENCES

[1] International Telecommunications Union (2011). "ICT Facts and Figures - The World in 2011." Available at http://www.itu.int/ITU-D/ict/facts/2011/material/ICTFactsFigures2011.pdf

[2] CTIA - The Wireless Association. "Wireless Quick Facts - Year-End Figures." Accessed July 31, 2013 from http://www.ctia.org/advocacy/research/index.cfm/aid/10323

[3] Synovate. "Synovate mobile phones survey." Accessed January 31, 2010 from http://www.synovate.com/insights/infact/issues/200909/

[4] Kari Edison Watkins, Brian Ferris, Alan Borning, G. Scott Rutherford, and David Layton (2011), "Where Is My Bus? Impact of mobile real-time information on the perceived and actual wait time of transit riders," *Transportation Research Part A: Policy and Practice*, Vol. 45 pp. 839-848.

[5] Sean J. Barbeau, Nevine L. Georggi, and Philip L. Winters (2010), "Global Positioning System Integrated with Personalized Real-Time Transit Information from Automatic Vehicle Location," *Transportation Research Record: Journal of the Transportation Research Board*, pp. 168-176.

[6] Sean Barbeau, Rafael Perez, Miguel Labrador, Alfredo Perez, Philip Winters, and Nevine Georggi (2011), "A Location-Aware Framework for Intelligent Real-Time Mobile Applications," *Pervasive Computing, IEEE*, Vol. 10 pp. 58-67.

[7] Michael Frumin. "CEN TC278 WG3 SG7: SIRI Version 2.0." Accessed July 31, 2013 from https://groups.google.com/forum/#!topic/siri-developers/apb20g8_0EI

[8] Metropolitan Transportation Authority. "MTA Bus Time(R)." Accessed August 1, 2012 from http://bustime.mta.info/

[9] University of South Florida. "SIRI REST Client." Accessed July 31, 2013 from https://github.com/CUTR-at-USF/SiriRestClient/wiki

[10] Internet Engineering Task Force, "Request for Comments (RFC) 2616 - Hypertext Transfer Protocol -- HTTP/1.1," ed, 1999.

[11] World Wide Web Consortium (W3C), "SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)," ed, 2007.

[12] Ecma International, "Standard ECMA-262 3rd Edition," ed. Geneva, Switzerland: Ecma International,,, 2011.

[13] LLC FasterXML, ,. "Jackson JSON and XML Processor Wiki." Accessed July 31, 2013 from http://wiki.fasterxml.com/JacksonHome

[14] Martin Adamek. "JSON Parsers Performance on Android (With Warmup and Multiple Iterations)." Accessed July 31, 2013 from http://martinadamek.com/2011/02/04/json-parsers-performance-on-android-with-warmup-and-multiple-iterations/

[15] Tatu Saloranta. "JSON data binding performance (again!): Jackson / Google-gson / JSON Tools... and FlexJSON too." Accessed July 31, 2013 from http://www.cowtowncoder.com/blog/archives/2009/12/entry_345.html

[16] S. Barbeau. "Parsing JSON and XML on Android." Accessed July 31, 2013 from https://github.com/CUTR-at-USF/SiriRestClient/wiki/Parsing-JSON-and-XML-on-Android

[17] University of South Florida. "SIRI Rest Client UI." Accessed July 31, 2013 from https://github.com/CUTR-at-USF/SiriRestClientUI/wiki

[18] Android Open Source Project. "Issue 43827: AnnotationFactory's cache has major performance problems." Accessed July 31, 2013 from https://code.google.com/p/android/issues/detail?id=43827

[19] Eishay Smith. "JVM Serializers." Accessed November 15, 2013 from https://github.com/eishay/jvm-serializers/wiki

[20] R. Jota, A. Ng, P. Dietz, and D. Wigdor, "How Fast is Fast Enough? A Study of the Effects of Latency in Direct-Touch Pointing Tasks," presented at the CHI 2013, Paris, France, 2013.