Smarter Apps for Smarter Phones!

A guide to improve apps connectivity, power consumption, user experience, security, and device battery life.

Version 2.0

12 February 2013

*This is a **Non-binding Permanent Reference Document** of the GSMA*

Security Classification: Confidential - Full, Rapporteur, and Associate Members

Copyright Notice

Disclaimer

Antitrust Notice

To the developers:

Smartphones have changed the way information is accessed. They have catapulted the development and distribution of mobile apps to a new level.

However, unlike fixed networks, the mobile environment places constraints on the resources available to apps on the mobile device. For example, the power consumption of each application can have an extreme impact on battery life. The frequency of device-server communication needs to strike a balance between delivering a good user experience while not draining the battery or impacting the user's phone bill (e.g. when roaming). High traffic levels can cause signalling overload in the network, triggering delays that impact the app performance and user experience.

Understanding and applying key principles of the mobile environment will help you improve your app's connectivity, data and power consumption and security. This will improve the user experience, and help to create and maintain the popularity of your app.
This document explains key differences between fixed and mobile environments, and highlights key principles to bear in mind when developing applications for mobile devices. It also provides detailed tips for Android, Windows Phone and iOS.

The following table outlines key recommendations with detailed explanations in later chapters. Considering these will help to make your app even smarter.

High level recommendations:

Term Description:

Relevance	**Guideline**	**For more details**
Usability/ Asynchrony	Techniques such as pipelining and asynchrony should be used to ensure that the client operates smoothly	Sections 2.2.1, 4.1.1, 4.2.1, 4.3.1
Efficient network connection usage	Use strategies that minimise and optimise data traffic and avoid unnecessary data transfers, especially when roaming.	Section 2.3 Background/ foreground modes Deactivate background processes when not required. Section 2.3, 3.6, 4.2.8
Background/ foreground modes	Deactivate background processes when not required.	Section 2.3, 3.6, 4.2.8
Background/ foreground modes, Scheduling	Design polling applications to aggregate their network activities.	Section 2.3, 3.6, 4.2.9
Connection loss and error handling	Applications should be resilient to changing network conditions and errors.	Section 3.2
Compression	Applications using HTTP should support compression.	Section 3.5
Data push	Applications should use push services in preference to polling.	Section 3.6, 4.2.5, 4.3.3

These guidelines have been compiled with inputs from developers, operators and terminal vendors. Updated versions will be provided, enhancing the contents and extending the scope to other relevant technologies and platforms. Although the underlying focus of the guidelines are predicated on addressing relevant issues in the context of wireless connectivity, similar issues may require attention in a wider context. AQUA (App Quality

Alliance) has published its 'Best Practice Guidelines, AQuA Test criteria for Android' that complement the current document.

We want to continuously improve the content of this document. Should you wish to contribute, please contact us at devguide@gsma.com

Alternatively you can join the dedicated W3C community discussion at:
http://www.w3.org/community/networkfriendly/

Table of Contents

Note: The content of this developer guide will (soon) be made available online under: http://www.gsma.com/smarterappsguidelines for easy use and ability to feedback or enhance.

1 Introduction

1.1 Overview

The rapid rise in demand for mobile data has taken key industry stakeholders by surprise, particularly the network operators at the forefront of delivering services to customers. A direct consequence of the huge success in the uptake of data services is a greatly increased signalling load at the network level independent of the volume of data traffic. End-users and application developers are unaware of increased signalling load as this is only visible to network operators/service providers. However, increased signalling load impacts smartphone users, who can experience rapid battery drainage, unresponsive user interface, slow network access and non-functional applications.

As use of smartphone applications increases, so does the signalling load on a disproportionate scale. This is caused by a number of factors, but aspiring enthusiasts are one of the main culprits, (perhaps with a background in developing desktop applications) who are translating their ideas into network-unfriendly apps that can be easily installed on smartphones.

As a result, network operators are facing the challenge of unprecedented signalling load that is out of proportion to the level of data usage.

The industry has responded by introducing the ‘fast dormancy’ feature. This means the mobile device notifies the network that its data session is complete, and requests the device be moved to a more battery efficient state controlled by the network. This has been implemented in what is known as 3GPP release 8.

A number of other aspects relating to the development of network-friendly smartphone apps need to be considered. These include:

a) Optimal use of wireless connectivity on target platforms by third party developers
   - This leads to better data bandwidth usage

b) Competent development of third party apps that are user and network friendly
   - This provides a much improved user experience and can improve battery efficiency

c) Identifying and addressing underlying peculiarities in smartphone software platforms
   - This improves network performance, user friendliness and battery consumption

d) Robust handling of failures
   - This can reduce battery consumption and reduce unnecessary data bandwidth usage

1.2 Scope

This document is designed to provide as much information as possible to all developers (private application designers, operators or OEMs) to encourage a better approach in developing mobile apps.

By following the guidelines and recommendations, developers will be better equipped to create fit-for-purpose apps; mobile operators will see a reduced strain on mobile networks leading to more responsive and reliable apps and improved battery life.

Network efficient apps will benefit developers by:

- Improving the overall user experience of apps, making them more responsive, providing more control to users, and providing better user experience due to less loaded/congested networks
- Improving reliability in the mobile network environment
- Providing higher levels of user satisfaction by reducing traffic levels, potentially resulting in lower customer bills and improved device battery life

The scope of these developer guidelines is limited to:

- General guidelines for native apps that require mobile network connectivity
- Specific guidelines for iOS, Android and Windows Phone. These specific guidelines should be updated periodically as target platforms evolve over time

The theoretical parts of Sections 3 and 4 are generic; they can be applied to any other platforms.

This document does not provide guidelines on:

- Generic user interface
- Complete device security, it only highlights what is available per platform
- Back end implementation
- The higher levels of security required in rare cases for specific apps serving banking or enterprise systems
- Web applications (HTML 5): Relevant developer guidelines have already been published on 14 December 2010 by W3C (Mobile Web Application Best Practices) http://www.w3.org/TR/2010/REC-mwabp-20101214/
- M2M (Machine to Machine)

1.2.1 Who should read this document

The document is not meant to explain the basics of developing a mobile app. It is aimed at developers (private application designers, operators or OEMs) who are able to develop or intend to develop mobile network-dependent apps. The “Detailed Recommendations” in Chapter 4 are aimed at improving the quality of apps relying on mobile network connectivity, and explain how to overcome the challenges that mobile networks introduce.

1.2.2 Organisation of the document

Chapter 2 provides the relevant background and lays down the fundamental constraints that are generic to all mobile platforms.

Chapter 3 considers the characteristics of an ‘ideal’ app/platform, to demonstrate optimal use of network connectivity.

Chapter 4 maps the outcome of preceding chapters to target platforms, highlighting specific functionality or limitations to further assist developers.

1.3 Definition of Terms

Term	Description
3G	3G is short for 3rd Generation, and usually to refer to mobile networks offering data rates over 200kbit/s
3GPP	The 3rd Generation Partnership Project (3GPP) is a global telecomm organization that defines and maintains standards and recommendations for the deployment of the GSM family of network.

APIs	Application Programming Interface (API) is a source code based specification intended to be used as an interface by software components to communicate with each other. An API may include specifications for routines, data structures, object classes, and variables
EDGE	Enhanced Data rates for GSM Evolution (EDGE) is a mobile network technology offering download speeds up to 236.8kbit/s
FACH	Forward Access Channel (FACH) is a radio channel used in UMTS networks that provides limited connectivity with battery drain than a dedicated radio connection
FIN	FIN is a finish message; it is a TCP segment with the FIN bit set, indicating that a device wants to terminate the connection
GPRS	General Packet Radio Service (GPRS) is a mobile network technology offering download speeds up to 60kbit/s
JSON	JavaScript Object Notation, is a lightweight text-based open standard, designed for human-readable data interchange. It is derived from the JavaScript scripting language for representing simple data structures and associative arrays, called objects

2 Network friendliness

Today's mobile broadband downlink speeds can range from 1.8 Mb/s upwards.

In contrast, fixed line broadband based on cable-modem or ADSL/DSL technologies provides a connection speed of up to 50Mb/s downlink. Fixed line broadband deploys less complex technologies than mobile broadband, and Wi-Fi offers very limited terminal mobility. Mobile networks differ from fixed broadband networks in that they have limited variable bandwidth, higher latency and a non-permanent communication channel. Loss of Internet is not considered abnormal.

Mobile networks have their own specific requirements and constraints, and even a Wi-Fi connection may not deliver the steady connectivity of the fixed network. As a developer, you should take these into account as you design and build your apps. These requirements and constraints are described in the following section.

2.1 Requirements and constraints in mobile broadband

- *Limited bandwidth*: The available bandwidth for mobile networks may vary depending on the geographic coverage and the underlying technologies used. On average it is lower than a Wi-Fi connection. In addition, when the mobile consumer is on the move, the bandwidth can dynamically step up or down
- *Data is not always free*: Outside monthly allocations and bundled price plans, mobile data usage can be expensive particularly when roaming. This can mean high bills for users
- *Battery life*: Mobile terminals are a miniaturised feast of technologies powered by a battery. When in full operation, the battery runs a processor with an active screen and data communication over the mobile network. Transferring large amounts of data puts the radio access into high drive mode. Add an active colour screen and the battery can drain in just a few hours. Considered use of the network, screen and processor resources when designing an app can dramatically improve battery life. For example, serving ads is popular with free apps but it can dramatically impact battery life and bandwidth usage. This could be improved by reducing the number/frequency of different ads being downloaded, or by introducing an ad-free (often paid) version that doesn't contain ads. *Network connectivity*: Mobile networks cannot by nature guarantee reliable connectivity at all times. Blind coverage spots, the limitations of deployed technologies, switching between cells, or moving in heavily built-up areas,

can all result in lost data packets, increased latency, reduced network speed, and connectivity interruption

- *Security*: Users do not always have direct control over their choice of wireless access networks. They can be connected to public Wi-Fi hotspots or in extreme cases even to spoofed networks, so privacy can be compromised or identity stolen. Authentication, secure protocols and a cautious approach to content transmission should be adopted by all developers

When network communication is optimised, the overall user experience is greatly improved. Developers should adopt all possible methods of optimal data transmission (efficient protocols, caching, compression, data aggregation, pipelining, etc.).

Although many mobile users have access to Wi-Fi networks at home, work or public places, their primary access to the Internet is via the mobile network. Developers often do not take this into account and do not perform rigorous field testing in the mobile environment – hoping instead that users will find a reliable connection. Development in simulated environments running on fast and well-connected laboratory machines may never uncover real-life user experiences. Therefore day-to-day testing of your app on a device connected to a commercial mobile network is essential.

2.2 Smooth user experience

Although network efficiency may be understood as the most effective use of bandwidth, it is also important to pay attention to the reality of mobile devices and mobile networks. All users today know that a mobile connection can be lost or data transfer delayed. The user experience of network friendly apps should be adjusted accordingly to smooth the impact of such issues.

2.2.1 Asynchrony

The first assumption to be made is that any response in a mobile network environment might be delayed or not delivered at all. To ensure a smooth user experience, an app's architecture should not solely rely on a sequence of responses, but be ready to deliver some results to the user even if not all the data has arrived.

A basic item list explains the problem in general terms. Figure 1 shows the sequence of requests required to download if all requests had been made synchronously:

Figure 1: Synchronous requests

In this example the list contains three items.

If the same requests were sent in parallel, then the timeline will be as shown in Figure 2:

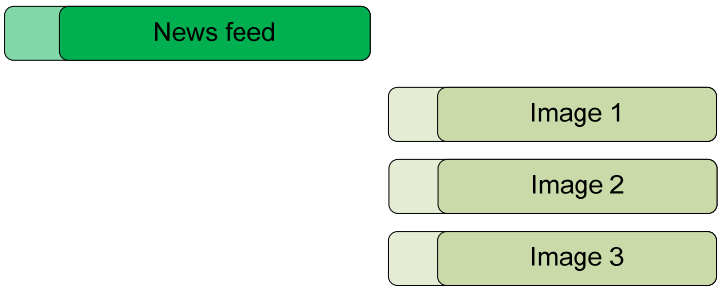

Figure 2: Asynchronous requests

Should the network connection be reliable with constant speed, the user will not notice the requests had been sent in parallel. The overall loading time will not show a tangible difference. However, such an arrangement can only exist in 'ideal' networks, with no latencies and connection interruptions.

In reality, the same sequence could potentially result in the arrangement in Figure 3, where a requested image may be received much later and some requests might not receive any response at all.

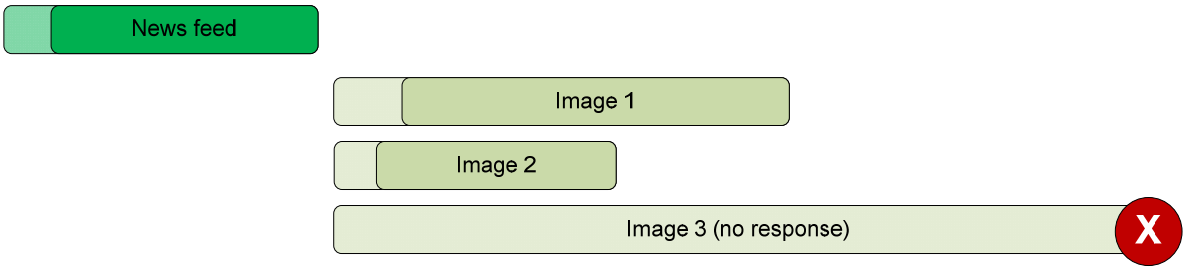

Figure 3: Asynchronous request in in reality

If an app waits to receive every single response and does not progressively show results to the user before completion of the entire cycle (as described above), the user might simply face a blank screen.
Network connections should be arranged in an asynchronous manner. This separation will ensure that delayed responses will not block the user interaction entirely.
Where possible, the user should be able to see the progress of data loading. This could be achieved by using progress bars, placeholders or a simple network indicator. In Figure 4, text information can be displayed already when the list is loaded without waiting for images to arrive. As soon as an image is loaded it can be displayed immediately.

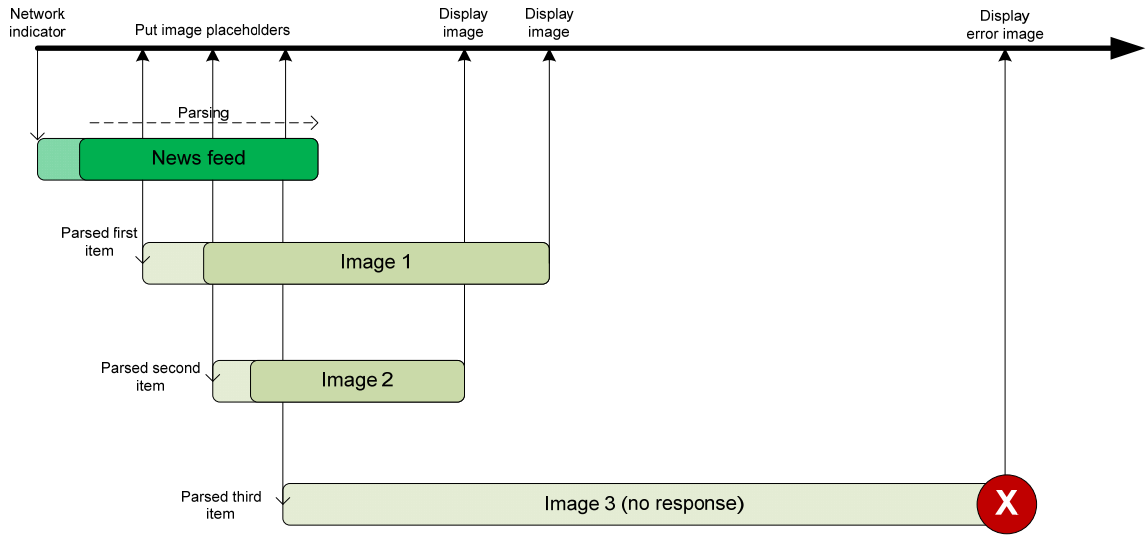

Figure 4: Timeline of asynchronous request

Apps should assume that any of the requested responses may fail to arrive. An appropriate user interface should keep the user informed of the progress without giving the impression that the software has crashed or hung.

2.2.2 Non-blocking user interface

A blocking User Interface (UI) is where the user is faced with a single UI element that prevents use of the mobile device. These can pop up from an app if there is a delay in receiving data, or when the app logic's decision tree is unable to proceed because it has encountered a missing data item.

In reality it is not necessary for an app to block the user from other operations. Even during a login process, when a user cannot progress any further within that app until access is granted, it should be possible to use other device applications.

In most cases, network operation should be completed in the background, allowing the user to cancel or switch to other views. It is inconvenient to the user to allow a web browser to block the screen with the message "Loading" until the page completes.

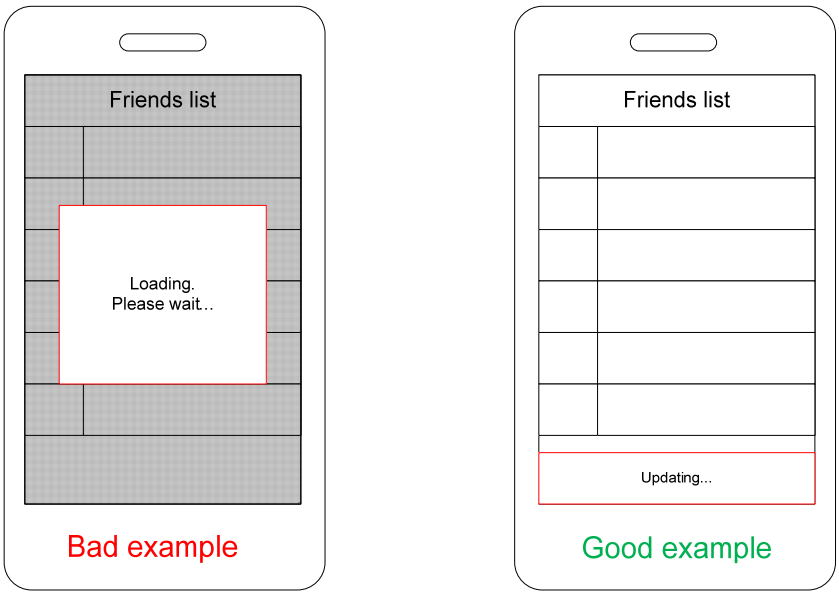

Figure 5: Non-blocking user interface examples

When designing an app's UI and its decision tree it is important to distinguish between a user-initiated network connection and an application-driven activity. This can define how the user is notified of progress and errors.

For example, if the user requests a web page to be loaded and the browser fails to connect to the server, then a modal error message (dialogue/information box) should be displayed. However, if an image has not been delivered, it would be more sensible to show placeholders with broken images instead.

Another example of an unhelpful error message occurs in some offline games. Whenever these games are launched on an unconnected device, an error message is often shown that reads "Could not connect to server", probably as a result of failure to send game statistics back to the server. The user is not expecting any result from a server, and these irrelevant messages can create an unnecessary and annoying break in the user experience.

2.2.3 Offline mode

There are occasions when a mobile device cannot connect or remain connected to the network, so it is important that developers take the following into consideration when building an app:

- If the network connection drops, the user should be alerted as to why an operation could not be completed
- To prevent data loss, users should be able to save current or active data with the option to retry/resume the activity when reconnected to the network

Examples of user disappointment include losing a long text string typed on a mobile device keyboard when it should be clear that the application cannot send the text to the server; or after downloading a huge chunk of data, finding it impossible to resume downloading and having to start the whole process all over again

- The user should be notified of any functionality that is not available in offline mode
- It is best practice to enable continued use of an app with data stored in offline mode for later synchronisation when the network connection is re-established
- The app should be capable of scanning for data connectivity in background mode without affecting operation in offline mode

2.2.4 Bandwidth awareness

Apps with excessive network dependency, such as audio or video streaming, require an assured level of data transmission speed. Considering the variety of wireless technologies such as GPRS, EDGE, 3G or Wi-Fi, it would be sensible for the app to first ascertain the access network and connection quality in order to request the appropriate quality of content from the server; and notify the user about the possible additional cost of using mobile data. If the app needs a more precise estimation of speed, then it would be reasonable to measure or dynamically adjust the quality of streamed data according to latencies.

The app should be capable of adapting to changes in access network and data speed at any given time, and make allowances for users leaving a Wi-Fi Hotspot, for example, or a mobile network handover from 3G to GPRS.

2.3 Efficient network connection usage

The constraints and limitations of wireless technologies have already been highlighted. Operating within these limitations means the frugal use of any data upload/download that impacts a user's mobile data plan charges when roaming, user experience responsiveness, and device battery life. Any optimisation of traffic will be appreciated by users, so double check if all network transfers are really necessary, protocols are chosen optimally, and caching is used appropriately.

Apart from data traffic, there are a few behaviours in a 3G network that need additional consideration. These are caused by the implementation of Fast dormancy, a feature that aims to minimise network signalling and battery consumption, both key issues given the increasing number of smartphones and online applications.

When a device requests data to be sent or received over a mobile network, the device switches from an idle to a dedicated channel state that consumes about 60-100 times more power compared to the idle mode. However, the very process of switching requires sending network signalling messages that also take a certain amount of time. Keeping the device in a high power state is not an ideal option as the battery will drain rapidly.

Between the idle and dedicated channel states there are few more 3GPP radio resource control (RRC) states that come into use. Fast dormancy technology defines an algorithm that dictates when a device can be switched to lower state after the last data transmission. Figure 6 below shows how the power drops after a certain period of inactivity in data transfer. Times T1 and T2 are network dependent.

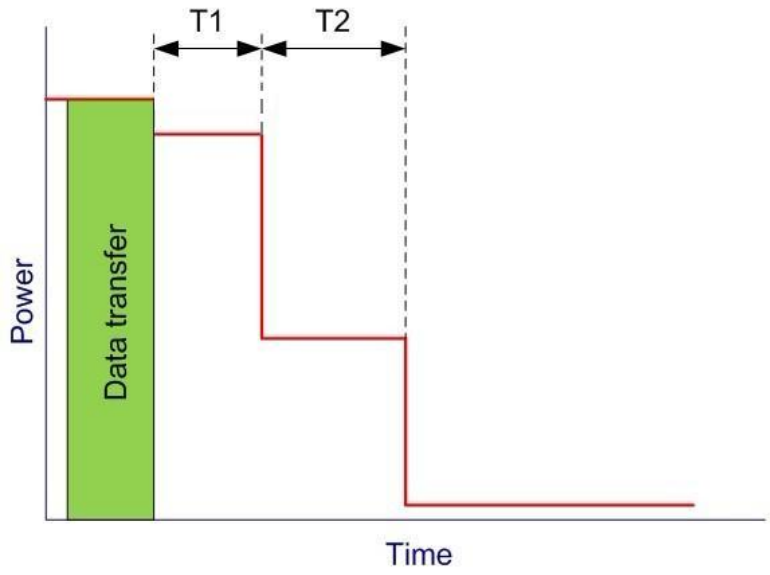

Figure 6: Power Consumption – Example 1

Once the state has switched to idle, establishing a new data connection may require the exchange of between 24-28 signals with the network, which could take one to two seconds.

This is an example of when the app has many short connections over a specific period of time:

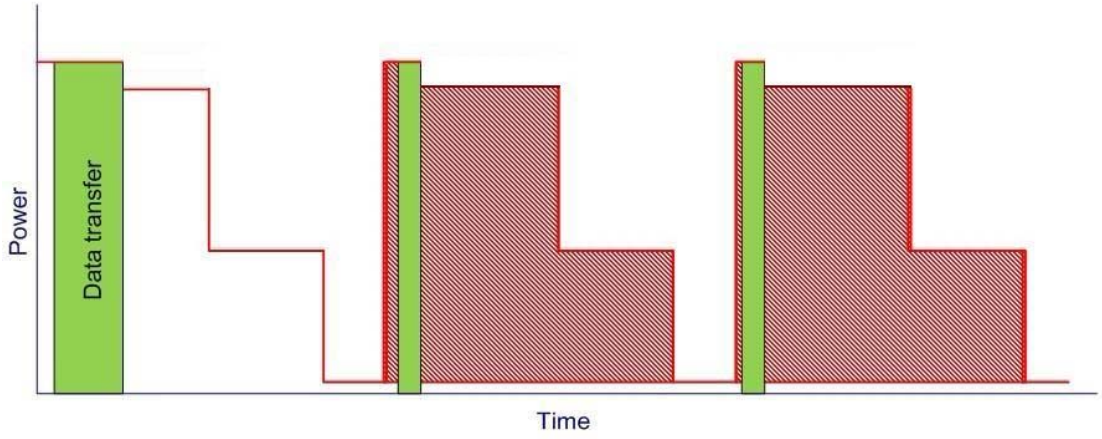

Figure 7: Power Consumption – Example 2

The red-hatched areas in Figure 7 show the overhead in battery usage compared to Figure 8 when all data connections are synchronised and completed in the same time.

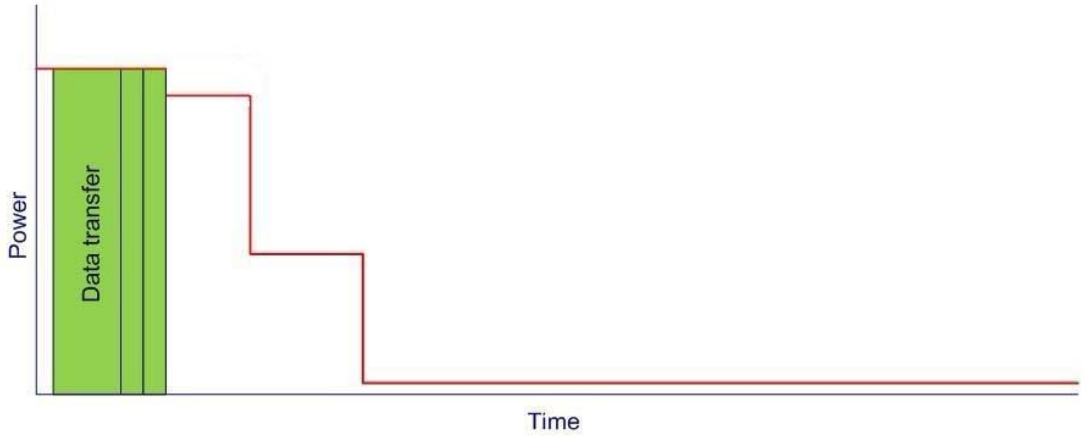

Figure 8: Power Consumption – Example 3

Although most the timers and conditions of switching between the channel states are network dependent, it is good to at least have an example of the approximate characteristics.

According to tests that have been done by XMPP Foundation:

Dedicated channel (the highest level) consumes about 380mA which can drain an average smartphone battery in less than four hours. The time before dropping to the lower state is approximately eight seconds

FACH (shared channel – intermediate level) consumes about 140mA. In order to keep this state and prevent switching into the higher power mode, the packet sizes must be around 128 bytes and after deducting TCP and TLS overheads this leaves only about 70 bytes of actual data. Timeout before switching to the lower state is around eight seconds. Battery life can reach a maximum of around seven hours in this mode.

The general recommendation is to transfer data in one go and not spread network activities. This should be done across multiple apps where possible and within apps (see 2.2.1).

In the 'across apps' scenario, the available scheduling mechanisms of the OS or the target application framework should be used. These are meant to ensure that the app's network activities, such as HTTP requests, are synchronised with other applications to achieve the behaviour explained in Figure 8 (for an example, see 4.2.9 for details on scheduling in the case of Android).

The same principle applies to push notifications too. Unless your app has real-time requirements you should not push notifications more often than you would have polled (sent a request to see if new data is available), if push was not available.

References XMPP on Mobile Devices: http://xmpp.org/extensions/xep-0286.html#sect-id115219

3 Ideal mobile application

We have already established the type of constraints that mobile apps need to address, where critical resources (such as battery, memory and processor) have certain limits.

Key generic characteristics of functionality or user case scenarios are addressed in subsequent sections.

3.1 Asynchrony

The concept of asynchrony has already been introduced briefly in chapter 2. There are two main aspects to asynchronous network connections:

- Network connections should not block the main thread responsible for handling user interface and system events
- If network requests do not depend on each other, they should be handled in parallel
- Asynchronous networking would always imply separate threads; although it makes the tracking of results and the state of an app non-trivial. This drawback, however, is well understood and competent solutions provided.

App architecture is driven by the APIs that platform vendors provide. To a great extent, the quality of most app implementations is dependent on the platform vendor's level of generic API support and optimisation at a platform level. For example, creating separate threads and managing them effectively should already be part of the underlying features of a target platform. This can save you time and money as you don't need to re-invent the wheel.

In this context the ideal APIs should have the following features:

- Creation and management of the network connections can be done from the main thread; however, the calls can lead to separate threads that are managed by framework transparent to the user
- All changes of states, received data, errors and timeouts are event driven
- The connection can be cancelled at any time

The design of APIs allows the simple management of several connections at the same time Developers are recommended to establish connections within a single connectivity session whenever it is possible to avoid losing dedicated channel state, which is described in Section 2.3. This reduces network signalling and, depending on the communication pattern, can make a significant impact on device battery life.

3.2 Connection loss and error handling

Monitoring connectivity status and error handling are extremely important as mobile networks are by definition not in a constant state.

Most platforms provide information on current connections. It is essential to check if the device is actually connected. Sometimes it is necessary to identify the type of connection: mobile network or, for example, Wi-Fi.

Although the actual bandwidth cannot be predicted precisely (as it depends on many factors, like signal strength, current network load, etc.), developers may assume that:

- Wi-Fi networks are generally faster than mobile networks
- Traffic over Wi-Fi is relatively cheaper in comparison, or free

If checks show the device is not connected, the app can switch to offline mode and let the user work with cached data only. This avoids handling inevitable network exceptions and notifications for each network error; the overall user experience is much smoother if constant error messages can be avoided. However, if the app switches into offline mode, it is best practice to monitor the device connectivity status so the app can switch back into online mode once a connection is established. At this point, data synchronisation between the server and client can be initiated or resumed.

Request types

When establishing the connection, different approaches can be used to display the status to the user and determine how to handle any network issues. A network request can be identified as user initiated if it is going to deliver the main information requested by the user. User initiated network requests can also be considered as primary.

Non-user initiated requests are those created by scheduled activities or triggered by a change in a system state, such as geo-location tracking or sending usage statistics to a server.

Secondary requests usually occur as a result of the primary request and do not bring any critical information to the user. Examples of secondary requests could be an image in a friends list (the list of names is critical), style sheets or images in web page.

Cancellation

Ideally, the user should see the progress of a primary request. It is also sensible to make the primary request cancellable, but this depends on the nature of the content and how it displays in the UI.

As a general rule if it is possible to perform any other operations on the same UI screen, it is a good practice to ensure ‘cancel’ is available as an option.

A good example when cancellation improves usability is the web browser, which is just another network-enabled application. A user can load different web pages on the same screen, so if the loading of one page takes too long, or there is a mistake in entering the address, the user can cancel the request and open a different web site.

When the primary request is cancelled, all secondary requests should be cancelled automatically.

Error handling

Mobile apps should always be prepared to handle situations when network requests fail. Most secondary requests can fail without a major impact on the user experience. Sometimes it is appropriate to indicate subtly in the UI that information for a secondary request cannot be delivered, such as broken image placeholders in web browsers or silhouette images in a contacts list.

When a primary request fails, it means that the main functionality cannot be completed and this is where error handling becomes important for the user experience.

As proposed earlier, it makes sense to distinguish between a user initiated request and non-user initiated (scheduled). If the request was user initiated and the information is expected to be delivered rapidly, then a modal error notification such as 'Retry' or 'Retry later' is appropriate. If a request is supposed to take longer time, and the user expects delivery to be guaranteed, for example, downloading music, an electronic book or a digital issue of a magazine, then in case of network failure, the app can automatically try to re-establish the connection. If up to five attempts have failed, then the request can be suspended (but not cancelled) with an option for manual resume later. It is also important to not lose any downloaded data and to be able to resume the download from the place where it has stopped rather than starting from scratch.

Retry mechanisms can vary and depend on the importance and volume of downloaded data. Possible solutions can be:

- Simple counting of failed attempts since the connection was first established (often the easiest solution).
- The number of failed attempts within a certain period of time.

For example, if the connection is lost more than five times within an hour, then the request can be suspended. This can be a more reliable technique to avoid short but regular network problems, such as when a device is moving away from one network cell to another. The connection can be lost when the device switches between cells, but when the cell is providing good coverage; the request can be processed successfully.

Regardless of the mechanism chosen, it is important to ensure that a failed operation will only be retried a limited number of timers. Without such a limit, an application may retry a failed operation for days or weeks while running in the background incurring data bandwidth usage and battery drain.

If the request is not user initiated then error notification can be either non-modal with a retry option or not shown at all. However, if the request is scheduled and repetitive, then it would make sense to change the interval dynamically to avoid re-establishing connections too frequently during network loss over a long period of time. Recommended retry intervals are one minute, then five minutes, and then 15 minutes. More frequent retries will drain the battery rapidly.

Resuming large downloads

The HTTP protocol supports requesting parts of files that can be used for resuming downloads. If the server supports it and the content can be returned split (i.e. content is not dynamic), then the server may include HTTP Header as described in sections 14.5 and 3.12 of RFC2616:

```
Accept-Ranges: bytes
```

The client can send subsequent requests for part of the file, specifying the download, for example, download first 500 bytes

```
Range: bytes=0-499
```

Or for segment starting from 9500th byte:

```
```

```
Range: bytes=9500-
```

The response HTTP Status 206 (Partial Content) will show if the requested range is correct, otherwise, there will be status 416 (Requested range not satisfiable). See Section 14.35 of RFC2616 for more details.

Section 3.f below describes how the verification of cached version can be done in HTTP using an ETag (entity tag). It is also possible to retrieve partial content with preceding verification of the content version by the HTTP request header If-Range, as specified in Section 14.27 of RFC2616. The idea is that the value of the If-Range header should contain the ETag value and the same request should also have a Range header specifying what part of content is to be received if the ETag is valid. If the server verifies the ETag, then the partial content should be returned, otherwise, the full version of the updated content will be sent.

Though the client can also use a Range header with conditional headers such as If-Unmodified-Since or If-Match, if the condition fails then client should handle the HTTP status code and a new request for retrieving the updated content. The If-Range header can help to do this in a single request using either ETag or last modified date.

Support for resuming downloads is extremely important for large content transfers on mobile devices, especially with the growing number of tablet devices, where quality of content is relatively high for a big display size. For instance, a single issue of a digital magazine can be 200-400 Mb. It is not acceptable for the user to have to download the whole file again if the network fails after already downloading several hundred megabytes.

In summary:

1. Check connection availability.
2. In offline mode use cached data.

    a) For any outgoing request that includes user-entered data, the data should be saved locally and an attempt made to deliver to the server.
    b) If delivery of the request fails, then the user should be asked if the request should be retried or retried later (with permanent saving in case the application is terminated).

If the primary request is done in online mode, then a progress indicator should be used to keep the user notified.

a) If the primary request is supposed to take more than one minute and the user expects to get the result however long it takes (download application, song, new magazine issue, e-book, etc.), then automatic retry should be implemented.
b) If several consecutive retries have failed, then manual retry can be implemented
c) It is good to indicate the progress of secondary requests, however, failure of them is not important and can be displayed only as a special placeholder (broken image placeholder for instance).
d) If the request is user initiated then error notification can be modal.
e) For repetitive scheduled requests, the retry interval should increase dynamically during long periods with no network connectivity.
f) Applications often fail to determine whether or not the user has any credit remaining if on a PAYG tariff. Lack of credit is quite common, and a status that may last for some time, so the application should specifically avoid making

repeated requests as the returned error messages will clog up the network and may not reflect the reality of the issue.

Caching

Caching is using the most effective means of data storage or transfer. For network applications, especially in mobile networks, the cache becomes essential. However, there are a few common challenges to address in terms of overall reliability, and ensuring the delivery of up-to-date information to the user.

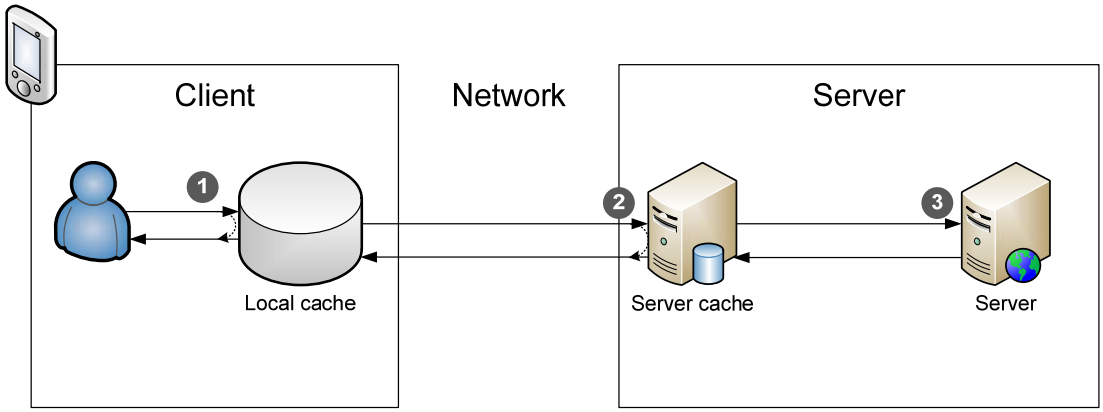

Figure 9: Caching

Although the entire client/server solution may contain many different levels of cache, generally two categories are supported: local cache and server cache. Local cache is used to minimise the number of network requests and enable faster delivery of results. The server cache works with the local cache to decrease the amount of data transferred via the network, whilst ensuring that the user gets the latest version of the information.

Figure 9 above shows the journey of a regular request from a mobile client to a web server:

- During the first stage the client checks if the requested content is stored in local cache and if it is still valid. If so, the data is sent to the user immediately without sending any requests to the network
- If the local cache contains data but needs validation, the client includes a version or checksum or the last modified date of the content that client already has. If the server cannot find a newer version of the content, it notifies the client that the local version can be used without sending the whole file over the network
- If there is no local version of the file, or the data is not up-to-date, then the server sends the latest version over the network. With proprietary implementations (depends on the nature of the requested data), it might be possible to send only changes to the local version

When designing an app, it is best practice to define the types of content that will be used and specify the caching strategy accordingly:

- Content can be cached without further validation. For example, if content has a unique identifier and cannot be modified on the server side, such as static photos in user albums (usually new photos can be added or old photos deleted, but not modified)
- Content can be cached locally, but needs validation with the server. A good example is the user's profile or profile picture which usually does not change very often but occasionally may be updated.
- Content cannot be cached at all. Examples: audio streaming, chat, etc.

Depending on the privacy of the content and security of local storage, some cacheable content should not be kept on device.

Local caches face the following problems:

- **Size limitations** – Device storage is always limited and depending on the app or the data, the cache should be limited to the corresponding size. Sometimes, it may be worth giving the user an option to define cache size as it will improve the perceived speed of the app for the user
- **Invalidation of content** – Usually web content has expiration date that can be defined by the server; however, it also can be defined manually depending on the nature of the data
- **Prioritisation of content** – As storage is limited, eventually the cache will be full. New entries in the cache should replace old ones with lower priority. The cache storage may have different strategies for this – removing the least frequent used, the oldest or the biggest entries

With HTTP version 1.1 the cache control became part of the standard and is well described in section 14.9 of RFC2616: which sets out the options for defining if content can be cached, the expiry date and the versioning of the content.

The HTTP protocol defines a mechanism for checking if the client's cache has the same version as the server. If the server recognises that the client has the up-to-date version of the requested data, then the response will consist only of HTTP headers and the whole content is not sent which can considerably reduce the network traffic.

The general idea is that on the first request the server sends a response with an additional header that can indicate the version of the content. The second request already comes from the client with information about the version to the server and if the server does not have any updates to it, it replies with HTTP Status Code 304 (Not Modified), or, otherwise, it sends the full content with the new version indication.

The version can be indicated simply by the last modified date in the Last-Modified HTTP response header (See Section 14.29 of RFC2616 for more details). The consequent request should come with HTTP request header "If-Modified-Since", as defined in Section 14.25 of RFC2616 or "If-Unmodified-Since", as defined in Section 14.28 of RFC2616.

Example

First request:

```
GET /image.png HTTP/1.1
Host: www.example.com
Connection: keep-alive
```

First response:

```
HTTP/1.1 200 OK
Cache-Control: max-age=31536000
Content-Type: image/png
Date: Mon, 21 Feb 2011 12:41:47 GMT
Expires: Tue, 21 Feb 2012 12:41:47 GMT
ETag: "11f-49bc3eabc9c80"
Last-Modified: Tue, 08 Feb 2011 11:47:46 GMT
Content-Length: 28702
Connection: Keep-Alive
```

Consequent request:

```
GET /image.png HTTP/1.1
Host: www.example.com
If-Modified-Since: Tue, 08 Feb 2011 11:47:46 GMT
Connection: keep-alive
```

Response:

```
HTTP/1.1 304 Not Modified
Date: Mon, 21 Feb 2011 12:44:07 GMT
```

This example shows that consequent requests can produce huge savings. In this case the response is short headers that are less than 1KB rather than 28KB of actual content, and reliability in delivering up-to-date content. If the server had a more recent copy of the picture, it would reply with 200 status and the full content instead of 304 HTTP status.

Content can also be marked with an ETag (see Section 3.11 of RFC2616) and these must be unique across all versions of all entities associated with a particular resource.

When the ETag is received from the server, then the client can use HTTP request headers:

- "If-Match" [RFC2616 section 14.24] – to deliver only the version that is requested, otherwise HTTP Status Code 412 (Precondition Failed) is returned
- "If-None-Match" [RFC2616 section 14.26] – to deliver only if the server has any other versions other than the client has, otherwise HTTP Status Code 304 (Not Modified)
- And "If-Range" [RFC2616 section 14.27] – to deliver part of file (using Range header) only if ETag matches, otherwise the whole file is delivered.

Taking the same example, the first response also includes the ETag, so the consequent requests either contain either only one condition or both conditions for the ETag and last modified date, for example:

Example :

Consequent request:

```
GET /image.png HTTP/1.1
Host: www.example.com
If-None-Match: "11f-49bc3eabc9c80"
Connection: keep-alive
```

Response:

```
HTTP/1.1 304 Not Modified
Date: Mon, 21 Feb 2011 12:44:07 GMT
```

When selecting a caching strategy, it is important for developers to evaluate the pros and cons of each mechanism, as differences in server implementations may have a significant impact on reliability and efficiency of the caching solution. Both Last-Modified-Since and ETag mechanisms have their own pros and cons, so bear in mind the following points:

- When the same content is distributed between multiple servers, unsynchronised time or an unsynchronised ETag generation algorithm can lead to inconsistent marking of

the content and therefore inconsistent responses from the servers. Server clusters or cloud based services are usually prone to such issues.

- For frequently changing or time sensitive contents (such as strongly related elements of the same data) preference should be given to the ETag mechanism, as it handles sub-second update issues.

3.3 Security

Although many aspects of security apply to both mobile apps and mobile platforms, this section addresses network security, covering secure data exchange between the mobile device and cloud web services. The key aspects are:

- Classification of information
- Authentication of users on web services
- Secure data exchange

The following aspects of security must always be taken into consideration by developers, but they are out of the scope of this document.

Device Security

Aspects of device access security, such as device unlock and remote wipe of storage in case of device loss

Content protection

Access control to user's personal data including personal contact information, address book, call history, SMS messages, mobile wallets, current location, passwords, VPN keys, etc.

- Encryption of locally stored data
- Protection against attacks
- Internal and external factors, damage caused by malicious software and viruses

Classification of information

When designing mobile apps, it is important to understand user concerns about data privacy.

In a simplistic way the data is classified as:

- Public: Information which is freely available on the Internet, can be found by other users, and cannot be associated with a particular user
- Private: The data which can be associated with an identifiable user, leading to compromised security

Below is an example list:

- Use case #1: The app provides read-only access to the information which can be easily found on the Internet by other users

  Classification: Public

- Use case #2: The app presents the same information as in Use Case #1, but some feedback is collected and stored in the cloud. This can be customer preferences, history of articles viewed, user comments or rating of the content.

  Classification: Private – as data associated with the user can be potentially used against him. The same data can be classified as Public if it is anonymised – this however, must be made clear to the user. User consent is required in both cases

- Use case #3: A productivity application, such as "TODO list", which synchronises data to the cloud.

  Classification: Private – the user could store sensitive information within the app, such as holiday dates, which can potentially indicate the location of the user. User consent is required

- Use case #4: A messaging or social networking app

  Classification: Private – the user can exchange sensitive information which could potentially compromise his security. User consent is required

When the data flow and sensitivity of transferred information is understood, it is a good time to estimate the impact on the user of monitoring ("Sniffing") of such traffic by an unauthorised party. Sniffing of user traffic may occur over Internet connections provided by public Wi-Fi access points, those provided by small businesses, or any other unregulated access point. It may take seconds for an intruder to intercept an authentication token and impersonate the user. A number of examples of such intrusions can be found on YouTube, including impersonation of users on social networks.

Authentication

Access to any Private data must be controlled and this is normally achieved by authenticating the client. The most basic authentication is achieved by validation of a pre-registered client ID with a password. Although client ID is most frequently just a personal email address of the user, device ID can also represent a client.

It is important to differentiate device authentication from device identification, where the latter does not require password validation and is often used by mobile network operators just to trace customers. Solutions relying on device identification pose a security threat if the mobile device contains or accesses Private data. Transfer of the mobile device to a different person if lost, stolen or sold, will automatically provide access to the data of the previous user.

Static device IDs such as serial number, telephone number or IMEI in clear form should never be used. Obscured device IDs (can be hash code based on the listed IDs) or automatically provisioned Unique Identifiers (UID) are acceptable and considered to be a good practice.

User authentication can also be implemented by integration into third-party authentication providers, such as Google ID, FaceBook ID or Microsoft Passport. For this reason, refer to APIs provided by these vendors or adopt open protocol OAuth (http://oauth.net).

Authentication must be performed every time the app establishes a new session.

Whichever approach is used for this purpose, it is important to ensure that:

- Authentication is performed using secure authentication protocols – Basic authentication over HTTPS is sufficient but over HTTP it is not enough. HTTP digest would be more appropriate, but again only becomes sufficiently secure over HTTPS. In some cases, a combination of stronger authentication over encrypted channel (SSL/TLS) is required. Proprietary implemented authentication must be performed over secure SSL/TLS based communication channel
- When a session is established, user or device credentials are not exchanged over an unsecured connection, so that session IDs, application PINs, service passwords, etc. are never exposed as these will provide an open door for intruders
- Apps should have an intelligent built-in logic to ensure all parameters related to user credentials (e.g. passwords, etc.) are populated prior to sending an authorisation request to the server

Strong Authentication

- Multi-factor authentication involves a combination of two or more stages. A variety of approaches exists – one example is a combination of user and server authentication, where verification of the server is performed by the client using additional security certificates. This type of authentication is used by businesses for implementation of Virtual Private Networks (VPNs).

Secure data exchange

- Implementation of secure communication using HTTP over SSL/TLS protocols (HTTPS) within the applications is not always favoured due to the effort involved. However, extra effort is needed for the implementation of secure solutions, and the investment you make in the security of your app will be recognised and appreciated by users. In many cases, the additional effort may be only the requirement to purchase and install a trusted certificate on the server and update the client to use HTTPS instead of HTTP.
- Encryption/decryption of traffic may have an impact on user experience, as additional processing time at both ends contributes to higher latency. This also has an impact on battery life. On high-end devices these drawbacks are addressed by hardware accelerated encryption, which maximises app performance.

Input Validation

The consequences of invalidated user input can be crashing apps, loss of data or theft of sensitive information as malware exploits breaches such as buffer overflow, format string vulnerabilities, stack overflow or race conditions.

Although many programming languages check input in standard APIs to prevent buffer overflows, native languages such as C, C++ and Objective C put this responsibility on the developer. Even though managed languages do aim for prevention, they still may be linked to native C libraries, and sometimes, open-source libraries that are not protected from defects and potential security problems.

Ideal platform

An ideal platform would:

- Support seamless secure user and server authentication
- Provide secure transport by default

- Provide secure storage for credentials

3.4 Efficient traffic usage

3.4.1 Cloud-based transformations

There is a category of mobile apps that use data from public resources such as news web sites. However, using public resources which are not under your control poses several risks as they fail to exploit standardised APIs, and are often inefficient:

- The format of data (HTML code) can be changed at any time which may cause app failure on the user's device.

The amount of data that is required for the app might be significantly more than actually necessary, thus increasing network traffic and latency.

In this case, it is highly recommended to check if there are any APIs (web services) provided from the public resource that are standardised, less likely to be changed and contain less unnecessary mark-up information.

Note that the API should not be used to deliver excessive amount of data to the app; otherwise its performance will decrease dramatically.

If no APIs are available, then you can also consider creating your own web service in order to have full control over the protocol and data being transferred between mobile device and server. In this case, even if the website changes its HTML code, then only the web service should be updated with the client remaining unchanged.

Many third party tools exist that can be used to transform content. A good example is Yahoo Pipes (http://pipes.yahoo.com). This provides a graphical user interface to aggregate, manipulate and mash up content from different sources around the web. Results can be delivered as RSS or JSON.

A few examples of types of operations that can be done with Yahoo Pipes:

- Fetch data from different sources like feeds, web pages, Google or Yahoo search, Flickr photos
- Custom input data can be used as an external parameter – i.e. a search query
- String manipulations such as regular expressions, text analyser, translation, etc.
- Location builder from a string
- Mathematical operations
- Filtering and sorting of the result

The picture below shows an example Yahoo Pipe that aggregates the results of search from four different sources, sorts the items by date and filters out non-unique titles, and compiles a result of maximum 40 news stories. It is also possible to combine the feeds of different languages and automatically translate them before aggregation

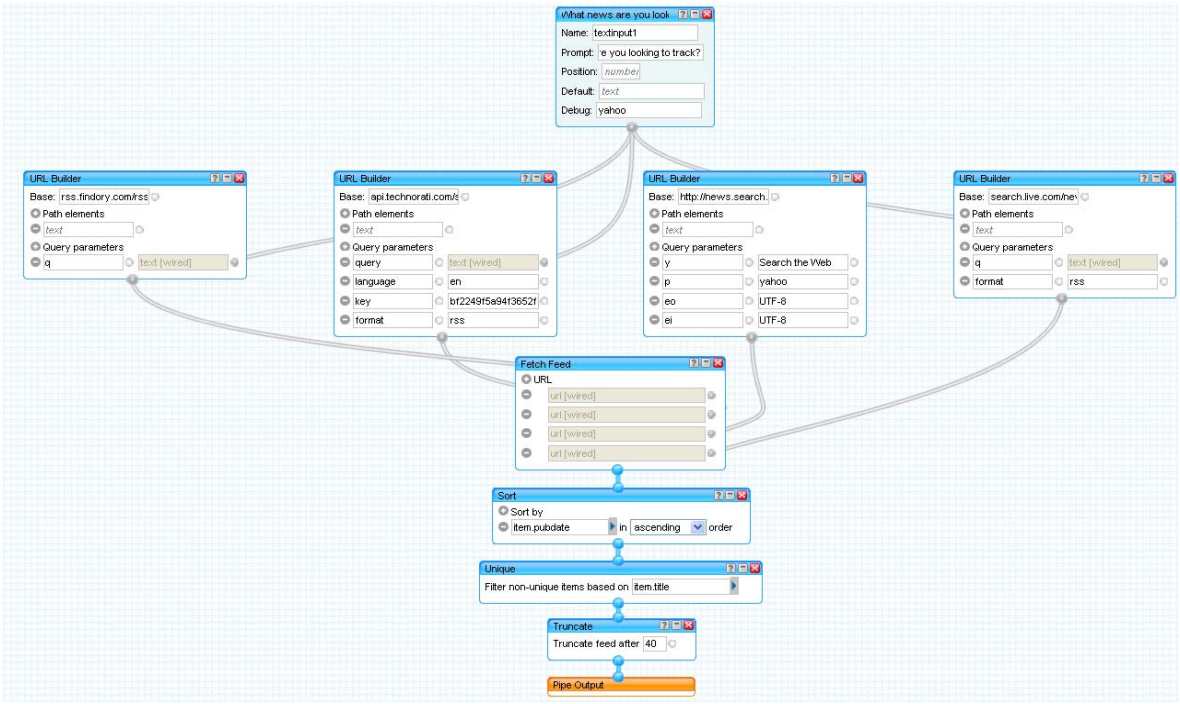

Figure 10: Yahoo Pipes solution

3.4.2 Media Transcoding

If the inefficiency in text based data formats can be improved by compression, the case for media formats – pictures, audio and video – is somewhat more complex as the quality of the media has a huge impact on its size. Therefore special care should be taken when transferring media.

Most mobile phones have fairly low (i.e. few-megapixel) cameras; however if an app uploads a picture taken by this camera for a social network website, which will reduce the size of any picture, there is no point in sending the image in its original quality. The difference in size can be around 30-50 times, which can also be the time difference taken in uploading the picture.

The same applies for downloading pictures. If the picture is supposed to be displayed only on the mobile device, then there is no point in downloading the original file size. This is always applicable, for example, for thumbnails; however, for full-screen photos some additional overhead might be allowed to allow users scaling up the image.

The size of video files can be enormous if a smartphone has an HD camera; in this case it might not be possible to upload a video over the mobile network without transcoding to a smaller, lower quality file.

If an app has video playback functionality, then a few points should be taken into account:

- It is better to not to exceed the resolution of the display where the video is going to be played (mobile device display or external display)

If the video is played in real time, then the bandwidth of the current network should be checked to identify the appropriate bit rate of video that can be played without constant delays. Progressive download and download resuming (section 3.2) may be used

Apple lays down strict requirements for online video in apps. If the video exceeds either 10 minutes duration or 5 Mb of data in a five minute period, you are required to use HTTP Live streaming; otherwise Progressive Download can be used.

As an alternative mechanism, MPEG DASH (Dynamic Adaptive Streaming over HTTP) provides a standardized, adaptive streaming protocol solution to use network resource efficiently.

Previously there were several commercial products for adaptive streaming such as Apple's HTTP Live streaming, MS's Smooth Streaming or Adobe's HTTP Dynamic Streaming. But In practice however, media service providers have adopted the streaming solution associated to the platform through which they delivered their media content, there was no interoperability between them. Acting on the demand from the industry, a standard for adaptive streaming, DASH was issued by MPEG.

More information about MPEG DASH can be found in <ISO/IEC 23009-1:2012>.

The ideal APIs on the platform to ensure that developers can leverage media transcoding would be:

- Basic image resizing
- Codecs that allow quality reduction (and size) of the audio file
- Reducing quality and resolution in order to reduce the size of the video file
- Support of media streaming protocols such as DASH, HTTP Live Streaming or RTSP

3.4.3 Presence

With the growth of presence-based services, it is important to manage high traffic and balance load generated by the services. Presence event distribution systems may generate numerous and unnecessary traffic such as separate presence subscription requests for multiple target users thus increasing the load on the mobile network. As such many common methods have been developed to reduce the network traffic generated by Presence event distribution systems.

Developers should consider applying such methods to their application to reduce the network traffic.

However, some optimization techniques may lengthen the delivery time of presence update, preventing users from receiving presence in a timely manner. For this reason, developers should also consider prioritizing presence information to be delivered when adopting some of these techniques

3.4.3.1 Bundling of individual presence subscription requests

Based on traditional mechanism, a presence subscription request is sent for each target user individually. When the number of target users to subscribe is large, Application developers should consider reducing the number of subscription requests by bundling them in a single request to reduce the network traffic generated.

For example, in SIP-based mechanism, RLS (Resource List Server) is a mechanism for subscribing to a list of target users. Instead of sending individual subscription requests, the watcher (requesting user) sends a single subscription request that contains a list of presentities (target users) to the RLS. Based on the list of presentities, the RLS sends multiple individual subscription requests to the presence server on behalf of the watcher. More detailed information about RLS can be found in <IETF RFC4662 A Session Initiation Protocol (SIP) Event Notification Extension for Resource Lists>.

3.4.3.2 Partial publication

Partial publication is a mechanism in which the target user sends only the parts of presence information that have changed since the previous update. Initially, complete presence information is sent to the watcher (requesting user) and then only parts of presence information are sent, reducing the amount of necessary data transferred over the network.

An example of mechanism for partial publication can be found in IETF draft-simple-partial-pidf format <IETF RFC 5262 Presence Information Data Format (PIDF) Extension for Partial Presence>. This technique does not reduce the number of presence notifications but reduces the size of the notifications. A watcher device can construct then the complete presence information from the partial publication received.

3.4.4 Email

Applications that send or receive email should carefully consider how to address a number of special cases specific to email:

a) Large message size

Many servers limit the maximum size of a message that can be sent. When sending large messages that may include pictures or video content, it is recommended that the application check the maximum message size supported by the server before sending the message to avoid wasting a large amount of data bandwidth and battery power transmitting a message that the server will not accept. This mechanism is described in RFC1870.
Similarly, when downloading messages it is recommended to check the message size before downloading it to avoid unnecessarily downloading email messages which may be arbitrarily large and may not be able to stored or displayed correctly on the device.

b) Frequent polling for messages

If possible, it is recommended to use a Push Server to notify the application when new messages are available on the server. If this is not supported, the client must periodically poll the server to check for new messages. Polling is very resource intensive on the device and frequent polling can have a significant impact on battery life. Developers are advised to carefully select the default and supported polling intervals in their application.
As a best practice, a default polling interval of 60-minutes is suggested with a minimum polling interval of no less than 15-minutes.

c) Error handling / retries

Special care needs to be taken when re-sending email messages that failed. In many cases, it is possible for the client to determine that the failure is permanent and the message can never be successfully sent – for example 5xx series errors in SMTP – these cases should never lead to a retry.
In cases where a network error or temporary server error occurred, the number of retries should be limited and staggered over time to limit the potential impact on data bandwidth and battery life. This is especially important if the client does not limit the maximum size of email messages since each attempt could lead to multiple MB of network traffic.

3.5 Compression

The HTTP protocol defines the mechanism of transferring data in compressed ways, if the server can support it, and most do. Enabling compression is a very simple task for the most popular web servers.

Compression can be very effective for XML or JSON formatted text data, by reducing the overall size by 80% on average. For binary contents, like photos or videos, however, compression does not make much difference.

The main idea of the HTTP compression is that if the client supports any of the standard compression methods such as GZip, Deflate (zlib) or LZW, then it mentions it in the request to the server. If the server supports the listed methods it can send a compressed response.

The indication that the client supports compressions is sent via HTTP Header Accept-Encoding.

Example request indicating that client supports GZip and Deflate compression methods:

```
GET / HTTP/1.1
Accept-Encoding: gzip, deflate
Host: www.example.com
```

Example response indicating that content is compressed:

```
HTTP/1.1 200 OK
Content-Length: 438
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip
…
…
```

RFC2616 section 14.3 and section 3.9 explain the Accept-Encoding header in more detail, and particularly, tips on defining the priorities (importance) of using different methods. The HTTP compression technique includes negotiation, to make sure that both the client and server support the same compression methods, so even more efficient methods can be implemented for certain types of content.

In order to simplify compression, the ideal API for HTTP client would support the main compression methods GZip, Deflate (zlib) and compress (LZW) with the corresponding Accept-Encoding header added by default and the content decompressed by default. However, you would also be able to disable or redefine handling of compression in order to support custom methods.

References Request compression

http://httpd.apache.org/docs/2.0/mod/mod_deflate.html#input

Speed Web delivery with HTTP compression

http://www.ibm.com/developerworks/web/library/wa-httpcomp/

RFC2616

http://www.ietf.org/rfc/rfc2616.txt

3.6 Background / Foreground modes

Most mobile platforms support some distinction between background and foreground modes for apps. The precise distinction varies from platform to platform but typically an app is said to be in the background if no part of its UI is visible and the user is unable to interact with it.

Given that a user interaction is not possible, careful consideration should be given to this aspect of app design to ensure that unnecessary network resources are not being used while in background mode. This will generally help to improve the user experience of the foreground application.

More specifically the app will receive some indication from the platform when a transition between modes occurs and should take advantage of this to gracefully release (or otherwise disable) the following application components:

- Handlers
- Timers
- Network transactions
- Memory/Objects
- Media codecs
- File & databases

Special attention should be given to apps that interact with the network on a regular basis as this drains the device battery and generates signalling traffic. In most cases the app should be prevented from interacting with the network whilst in background mode, as there is no way to present results, unless a notifications system is used. Idle screen widgets (e.g. weather / news) are common culprits here; however, this does not apply for apps such as instant messengers, as they need a constant connection.

There can be no hard and fast rules in this area – for instance a music player is likely to want to continue to decode audio even when in background mode. At the very least you should review the detailed operation of your app in each state to ensure its resource footprint is appropriate.

Similarly, apps that do need to interact with the network whilst in background mode should consider alternative approaches. For example it may be possible to batch the transactions of several apps so the background app can “piggy-back” its transactions. This batching capability may be provided by the platform itself; it is particularly important for background events when there is no user interaction with the phone (e.g. the phone is on a desk). A common reason for background apps to access the network is to poll a server, however a better approach is to use push notification (if supported).

The HTTP “Keep-Alive” mechanism is frequently used as the basis for push notification systems, but this only works well if there is a centralised client side component for receiving/routing notifications (i.e. as part of the platform e.g. Android GCM).

If push notifications are not available or not suitable, keep-alive connections can be used to replicate a push notification mechanism and avoid frequent polling of data. The main advantage of keep-alive over polling is that the connection can be kept open without frequent transfer of data, enabling the mobile state to switch to lower power. However, if anything needs to be delivered from the server, this can happen immediately. It is also necessary to make sure that the connection is still alive by sending non-frequent data packets (minimum 10 minutes, but slightly less than30 minutes seem to be the optimal setting as many firewall/NAT’s timeout TCP connection after 30-minutes).

Some platforms provide a richer (more fine grained) application lifecycle than others. You should exploit the lifecycle to its fullest to achieve the best user experience.

It may be desirable, for example, to retain a group of thumbnails across several application states that represent the “active” cycle of the app (where “active” might encompass background as well as foreground modes) but release them across states that represent a less active cycle. Failing to consider the target lifecycle can result in apps that perform well on one platform having poor performance on another.

You should also consider altering your app’s resource footprint in response to changes in the mobile device state. In some cases these may fall within the scope of the application lifecycle (e.g. an incoming phone call is likely to result in the app making a transition to background mode). Other changes may lead to a different form of notification (apps may need to register to receive screen lock event notifications for instance). A useful approach is to treat each state transition as a separate use-case and identify those cases that impact on the app. This would, for example, show that in the case of the music player mentioned earlier it might be worth shutting down the audio decoder task when the speaker is muted.

Another aspect to highlight is that many apps seem to trigger network activity (e.g. polling, status update) when the screen display turns on and the device wakes up from sleep mode. The intention may be to obtain/update the latest information when the user starts interacting with the device. However, the screen display may turn on regardless of the user's intention to interact with the device, and the device screen may remain locked. For example, the screen display may just turn on due alarm clocks or the user tapping the screen display to check time. Thus, to avoid unnecessary network signalling / draining of device battery, it is recommended to trigger such network activity when requested explicitly rather than being triggered by the event of screen display turning on.

3.7 Application Scaling

Your app should be designed to ensure that network activity is not concentrated at specific times and is tolerant of geographical loading problems

- Handsets are frequently synchronised to a standard clock source, so frequent updates using exact times (especially for apps that are used by many users) may cause short overloads to the application servers and the radio network. A better example of how to do this is Antivirus tools which launch update requests back to servers independently of one another. Perhaps the most popular example is RSS feed used in browser application. They may result in delayed responses and impacting user experience. Designing an app to spread network activity timing across different devices would reduce such overloads, and improve app performance and device battery life.
- To illustrate the point let us take a closer look at the RSS feed example (where it can also be implemented as a native application)
- RSS newsfeed may require the RSS reader on handsets to check for updates on servers periodically (e.g. every 30min), but not necessarily at exact times (e.g. XXhr:00min, XXhr:30min). In such cases, it would be ideal to evenly spread the network activity timings (i.e. the timings which the RSS readers checks for updates) across devices as in Figure 10a below.

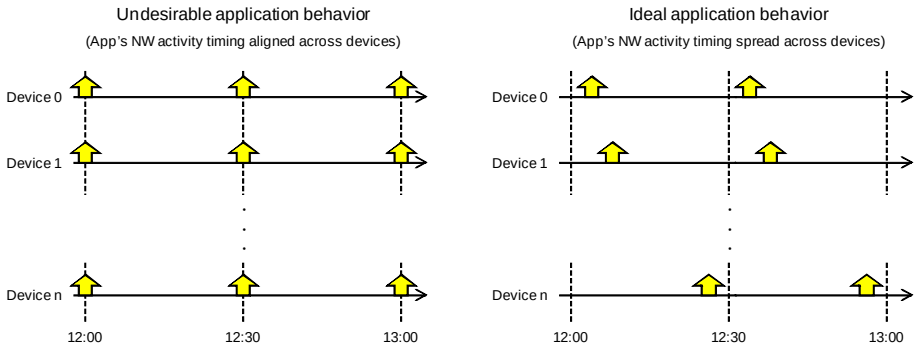

Figure 10a: Spreading an App's NW activity timing

- One way to realise such behaviour would be to schedule network activity timings using relative times (e.g. "30min from the current time"), and using a timing which would not be aligned across devices as the base timing. For example, the base timing can be the time when the device bootup.
- Weather widgets may require data retrieval from servers at exact times of a day (e.g. 05hr:00min, 11hr:00min, 17hr:00min) when the latest information is made available. In such cases, it would be better to spread the network activity timings (i.e. the

timings which the weather widgets retrieves data) across devices within an acceptable time window (e.g. 5min) as in Figure 10b below.

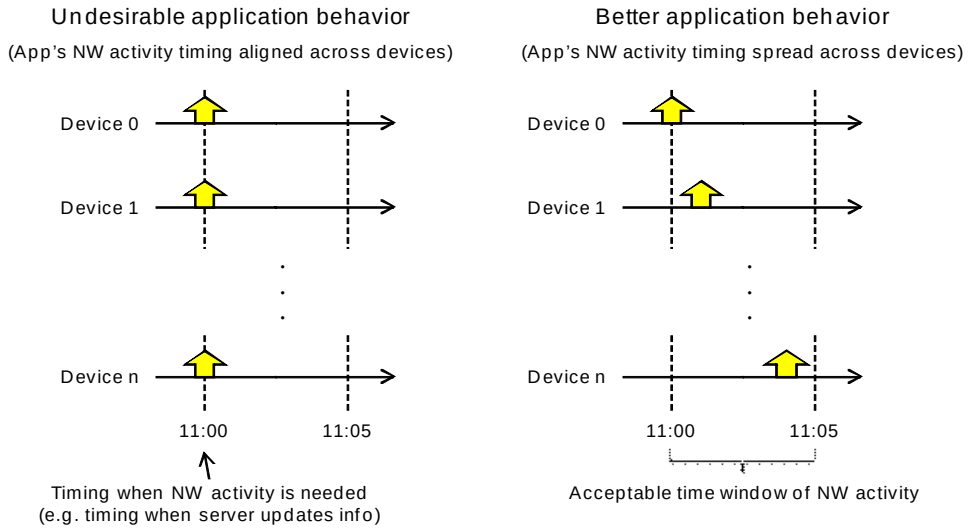

Figure 10b: Spreading an App's NW activity timing within an acceptable window

- Such behaviour can be realised by including a random offset (within a desired time window) when scheduling network activities. E.g. "Activity at 17hr:00min + offset", where the offset is defined with a random function having an uniform distribution within the desired window.
- Developers are recommended to avoid, as much as possible, using exact times for an app's network activities, and to use randomisation design techniques to spread network activity timings across different devices. The network capacity of a local area will be significantly lower than the product of the number of handsets and their assigned bandwidth. On occasions there may be large numbers of users in a specific location. In general, apps should use some randomisation design techniques to spread network synching and connectivity load.

4 Detailed Recommendations

4.1 iOS

4.1.1 Asynchrony

An app's main thread is responsible for all activities including handling of the system messages, input events, etc. iOS makes sure that the main thread is always alive by a mechanism called WatchDog. This can terminate the process if it does not respond within approximately 20 seconds. Therefore, if any synchronous operations are called, you need to be sure that these operations can be completed as fast as possible. This becomes critical in the mobile network environment, as network timeouts are much longer than the WatchDog's. For example, domain name resolution will be timed out after 30 seconds if there is no response from the network.

iOS APIs are designed to simplify development as much as possible, and in most cases you don't even need to think about creating separate threads, as everything is done transparently and asynchronously. However, some of the methods hide synchronous networking which should be used very carefully and only in separate threads. Apple provided a list of such methods in WWDC'10 which is:

- Utility methods:

  -initWithContentsOfUrl:

  +stringWithConten-tsOfURL:
- DNS:

  gethostbyname

  gethostbyaddr

  NSHost (Mac OS X)

  +sendSynchronousRequest:returningResponse:error:

As explained earlier, network asynchrony is not just calling network functions from the main thread to not block UI, but also handling requests and responses independently from each other. This can be done by using request queues and, although the standard iOS SDK does not provide this functionality, there are a few third party libraries that do.

iOS is a trademark or registered trademark of Cisco in the U.S. and other countries and is used under licence by Apple Inc.1

“Three20” library wraps Foundation’s classes such as NSURLRequest, NSURLConnection to add an extra functionality, for example, more advanced caching, queues and retry mechanism. The advantage of using the “Three20” network module can be significant if the network activities are integrated with any other “Three20” modules, especially with its user interface.

Another library that gives rich network functionality is ASIHTTPRequest. Unlike “Three20”, this library wraps lower level C API – CFNetwork, however, it is an Objective C library. Some developers may even prefer ASIHTTPRequest rather than standard Foundation API, as it implements many additional features such as:

- Request queue
- Simple API for sending POST requests with files attached and post values
- Tracking progress of a single request or the whole queue with automatic update of UIProgressView
- gzip compressed request bodies
- Resuming interrupted downloads Section 3.2
- Background-mode requests
- Client certificates support Section 3.4
- Automatic support of network indicator
- Automatic retry
- Persistent connections – to reuse single HTTP connections for a several small requests

References

ASIHTTPRequest documentation

http://allseeing-i.com/ASIHTTPRequest/How-to-use

Three20

http://three20.info/

4.1.2 Connection loss and error handling

As described in Section 3.2, it is good idea to make your app aware of the device connection status. If there is no connection, the app can switch to offline mode, avoiding network errors and protecting the user experience.

Apple provides sample code in the “Reachability” project (see reference below) which can be used for detecting network status and notifying changes. The method reachabilityForInternetConnection would be appropriate for most of the cases.

If the app detects that there is no Internet connection and offline mode should be used, then local data shall be used. The simplest solution without building a local database would be to use standard NSURLRequest, but with custom cache storage that supports on-disk cache (as described in Section 4.1.3) and cache policy parameter set to NSURLRequestReturnCacheDataDontLoad. This will avoid pointless attempts to establish a network connection, and will return the result immediately, if anything has been cached. The rest of the app can be left unchanged with error handling as if it was using the network.

For any network request that uploads data to a server regardless of size, for instance, uploading a picture, or updating status on a social network, it is advisable to use the Task Completion API to ensure content is delivered even if the user minimises the app. It is also important to ensure entered content is not lost if the network connection drops, and that the task can be retried without the need to re-enter the data or retake the picture.

Long downloads, such as music files, digital issues of magazines or any other large files, should be resumable. If the server and the content support HTTP partial download, the request for restoring the download from any part of the file can be initiated by adding the HTTP Range header:

```
// Requests part of file starting from 1024th byte
[URLRequest setValue: @"bytes=1024-" forHTTPHeaderField: @"Range"];
```

References

Reachability

http://developer.apple.com/library/ios/samplecode/Reachability/

Network reachability

http://blog.ddg.com/?p=24

4.1.3 Caching

The Foundation framework provides simple to use cache management, giving developers control over what can be cached and where. Standard NSURLCache is very limited, although the full Mac OS X2 version supports on-disk and on-memory caches, the iOS version can store only in memory. Furthermore, by default the capacity of memory cache gets set to zero, meaning that even memory cache will not work if it is not enabled explicitly.

A simple test shows that memory capacity is set to zero by WebView (even if it is not used in the UI) from a separate thread. To make the matters worse, the point at which this happens is not documented. A simple and reliable workaround is to subclass NSURLCache and redefine method setMemoryCapacity which will ignore all calls with value 0 and will pass through all other values to the original method.

Example

```
-(void)setMemoryCapacity:(NSUInteger)memoryCapacity {
    if (memoryCapacity == 0) {
        return;
    }
```

```
    [super setMemoryCapacity:memoryCapacity];
}
```

The standard class NSURLRequest includes a parameter cachePolicy that can retrieve the following values:

- NSURLRequestUseProtocolCachePolicy – the default cache policy for the protocol that is being used for the particular URL request. Suitable for most of the cases in online mode
- NSURLRequestRelaodIgnoringCacheData – ignores any local cache and will try to load data from the originating source. Suitable for online mode and for certain use cases, when data should not be cached, for example, for keep-alive connections
- NSURLRequestReturnCacheDataElseLoad – returns data from cache even if it is expired. If there is no cached version, then it will try to download the data from the originating source. Suitable for certain types of content such as static photos
- NSURLRequestReturnCacheDataDontLoad – returns data only if it has been stored in local cache and does not attempt to retrieve it from the origination source if there is no cached version. Suitable for offline mode

Example:

```
// Creating URL request
NSURLRequest          *theRequest          =          [NSURLRequest          requestWithUrl:
    [NSURL URLWithString:@"http://www.hudriks.com/example.html"]
    cachePolicy:NSUrlRequestUseProtocolCachePolicy
    timeoutInterval:60.0];

// Initiation the connection with the request
NSURLConnection          *theConnection          =          [[NSURLConnection          alloc]
    initWithRequest:theRequest delegate:self];

if (theConnection) {
    // Prepare for receiving the data
} else {
    // Handle the error
}
```

The framework also allows the response to be altered before it gets stored into local cache by implementing connection:willCacheResponse:. Usually, this method is used to avoid caching of some private data, and some implementations do not cache any traffic that goes through encrypted protocols such as HTTPS.

If in-memory cached is used, it would be reasonable to clean up the cached data if the application receives a memory warning.

Currently, the standard NSURLCache does not support all features of HTTP cache such as conditional HTTP request

headers, for instance, "If-None-Match", "If-Modified-Since", described in Section 3.3.

Although, the default NSURLCache is very limited and cannot help much for implementing offline mode, the API still gives a solid framework that can be used for simple integration of your own implementation of cache or subclass of NSURLCache class.

There are a few implementations of custom cache classes:

- "Three20" framework (see reference below), that has been developed for Facebook , also gives a choice of cache storage such as Memory, Disk or Network and provides separate in-memory storage specifically for images to optimise performance. However, their cache is not subclass of NSURLCache and requires using their request classes as well
- SDURLCache – subclass of NSURLCache with on-disk cache support

Cache in web applications

Key facts from Yahoo!'s research into how mobile Safari works on the iPhone regarding caching can be used for designing and implementing web applications:

http://yuiblog.com/blog/2008/02/06/iphone-cacheability/

In order to be cached by Safari, the HTTP content should include either "Expires" or "Cache-Control" header.

```
Expires: <Expiration time in GMT Format>
Cache-Control: max-age = <Expiration time in seconds>
```

The browser's cache applies a limit to the cacheable content, which should not be larger than 25 KB (or 15 KB according to the latest tests) of uncompressed data. Even if it is transferred using HTTP compression, the browser will still uncompress it before trying to put it into cache. This means, that the correct distribution of content over small files and the minimisation of each file, particularly, JavaScript, CSS and HTML, becomes highly important for the performance of mobile web applications.

References

Three20 Framework

https://github.com/facebook/three20

SDURLCache class

https://github.com/rs/SDURLCache

URL Loading System Programming Guide

https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/URLLoadingSystem/URLLoadingSystem.html

4.1.4 Security

Although the iOS operating system is based on Mac OS X and most of the security has been inherited from there because of the differences in the usage, there are some discrepancies in the APIs and security models.

iOS security is based on three main services in the Core Services layer, which are:

- Keychain Services – secure storage of passwords, keys, certificates and other secrets
- Certificate, Key and Trust Services – creating and managing certificates, creating encryption keys, encryption and decryption of data, signing and verification of digital signatures

- Randomisation Services – cryptographically secure pseudorandom numbers

On a higher level, CFNetwork and subsequently URL Loading System use these services, for instance for providing secure transport protocols and supporting SSL and HTTPS connections.

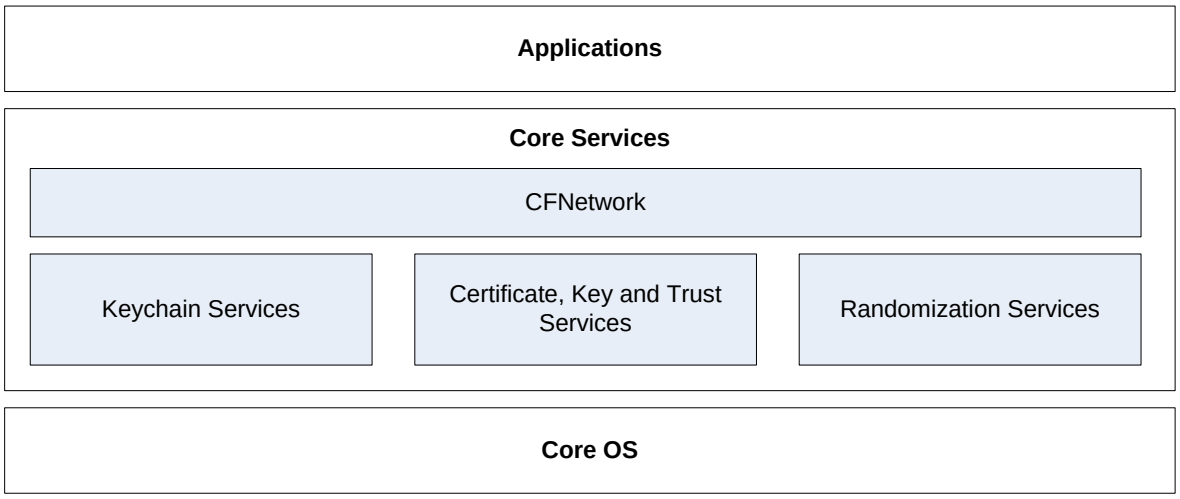

Figure 11: CFNetwork Component Diagram

Keychain Services in iOS has a major difference from the Mac OS X version. In Mac OS X, Keychain securely saves passwords and if an application requests information, the user is asked to give permission by entering his password.

In iOS the device is already secured by PIN number, and, therefore the user is not asked to enter any passwords or confirmations. However, Keychain allows access only to signed apps, each has individual storage and cannot access information from any other applications.

As mentioned earlier, URL Loading System supports the HTTPS protocol by default, so developers do not need to put any extra effort in establishing a secure connection with the server (if the server also supports HTTPS).

Apple has also done a great job in supporting developers and we highly recommend that you take note of the following documents from the Apple Developer Network:

- **Secure Coding Guide:** covers all aspects of security, not only network security. Explains in detail topics such as buffer overflow, stack overflow, input validation, how these can be used by attackers to run malicious code and how this can be avoided. The design of secure user interface is also touched on in the document which explains that security should not compromise usability of an app ¦
- **Security Overview**: gives more details about cryptography and secure APIs in iOS and Mac OS X
- **Keychain Services Programming Guide**: contains a section related to keychain services in iOS and gives guidance about how the relevant APIs should be used
- **Certificate, Key, and Trust Services Programming Guide:** Explains how the APIs for managing and using certificates and encryption/decryption of the data should be used.

4.1.5 Push notifications

Apple Push Notification service (APNs) is a robust and highly efficient service for propagating information from the cloud to iOS devices. Each device establishes an accredited and encrypted connection with the service and receives notifications over this persistent connection. Apps are notified about information waiting for them on their back end

servers, and expected to pull this information from the server. If the app is not running, notification is handled through the UI and alerts the user to launch the app.

More details on APNs are available at:

https://developer.apple.com/library/mac/#documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/Introduction/Introduction.html

4.1.6 Data formats

JSON

JSON is very popular these days and some web sites provide access to their APIs only using JSON rather than XML.

The most widely used Objective-C JSON parsers are YAJL, JSON Framework and Touch JSON (see references below). Each has its own advantages and disadvantages.

YAJL is sequential access parser which is similar to SAX parser for XML. As it does not need to keep all data in memory, the obvious advantages are low memory footprint and parsing speed which would be suitable for huge amounts of data or even streams of data.

The Touch JSON library shows good results in speed benchmark and it has a very simple to use API. For example, to parse JSON into NSDictionary object, the code looks like this:

```
SBJSON *jsonParser = [[SBJSON new] autorelease];
NSString *jsonString = ...;
return [jsonParser objectWithString:jsonString error:NULL];
```

It can be even simpler by using `NSString` extensions:

```
NSString *jsonString = ...;
return [jsonString JSONValue];
```

Encoding to JSON can be done by `NSObject` extension:

```
NSString *jsonString = ...;
NSDictionary *data = ...;

jsonString = [data JSONRepresentation];
```

XML

iOS SDK provides only the event-driven XML parser NSXMLParser which works in the same way as the SAX parser, but instead of callback functions it sends messages to its delegate:

- parser:didStartElement:namespaceURI:qualifiedName:attributes:
- parser:foundCharacters:
- parser:didEndElement:namespaceURI:qualifiedName:

There is also an alternative third party event-driven XML parser called AQXMLParser that gives considerable memory savings.

If the app needs a tree-based parser, despite memory consumption, it is possible to use the libxml2 library that is already included on the iPhone, however, it is a pure-C interface. The

other alternative might be using Objective-C Touch XML framework which is a wrapper for the libxml2 library.

As a rule of thumb, light-weight protocols should be used as they are much more suited to the mobile environment. A good example is using REST where possible instead of SOAP as REST protocols are much more suited to the mobile environment.

References

YAJL

http://github.com/gabriel/yajl-objc

JSON Framework

http://github.com/stig/json-framework

Touch JSON

http://github.com/schwa/TouchJSON

AQXMLParser

http://github.com/AlanQuatermain/aqtoolkit

Touch XML

http://github.com/schwa/TouchXML

4.1.7 Compression

iOS supports compression (gzip and deflate) by default and automatically adds “Accept-Encoding” header to all requests and then decompress the response. This increases the efficiency of data traffic.

ASIHTTPRequest library supports gzip only and “Three20” also supports gzip and deflate as it wraps the standard NSURLRequest.

4.1.8 Background / Foreground modes

With version 4, iOS started supporting multitasking on almost all devices apart from the iPhone 2G, iPhone 3G and corresponding models of the iPod Touch. However, the iOS version is not the same as the multitasking that developers are used to on desktop operating systems. The main difference is that iOS multitasking limits background activities due to the limited resources of the mobile device and the different usage of mobile apps.

iOS gives developers seven different background services that can be implemented in apps

- **Fast application switching** – suspending the app with preservation of its state and quick resume
- **Push notifications** – delivery of backend information to an app not currently running in foreground mode
- **Local notifications** – scheduling of delivery of push-style notifications, while an app is suspended or closed
- **Background audio** – playing audio content through the unified playback system on the device while an app is in background mode
- **Task completion** – gives extra time to complete a task in background
- **Location and navigation** – tracking the location changes
- **Voice over IP** – making and receiving calls using an Internet connection

Almost all these services may involve network activities (apart from fast application switching and local notifications), and extra care should be taken to not reduce device battery performance or overload the network.

Push notification is a well optimised technology compared to polling data. However, if not used carefully, it can cause problems in the network, mainly related to simultaneous broadcast of notifications to many devices (latest news, promotional offers, etc).

Other background services such as background audio, task completion, location and navigation and VoIP can be used for establishing frequent network connections, and can therefore drain the battery very quickly. The general advice here is to consider which data requires immediate delivery and which can be aggregated with its delivery postponed.

The Task Completion API gives some flexibility for developers to run almost any code, however, the intention is to give the app extra time to finish activities initiated while the app is in background mode. As soon as these are complete, the app can be suspended without using any resources.

This can be very useful for network operations that may take long time or require a certain level of reliability, for instance uploading pictures, or sending a text message/email. To start a connection in background mode, method beginBackgroundTaskWithExpirationHandler: in UIApplication should be called and when the activity is finished or it has failed, then endBackgroundTask: should be called iOS.

However, if the Task Completion API is not used, then the corresponding delegates for NSURLConnection are still called after resuming the app. If this happens within the network timeout (by default, 60 seconds), then the response may still be delivered, otherwise, delegate didFailWithError: is called.

Backward compatibility

As multitasking is not supported on the iPhone 3G and 2G and the iPod Touch 1st and 2nd generations, even if they have iOS 4 installed, it is important to check if an app can use the corresponding APIs on the device.

This can be done as follows:

```
[[UIDevice currentDevice] isMultitaskingSupported]
```

Or

```
if([someObject respondsToSelector: @selector(methodForMultitasking)) {
    ! [someObject methodForMultitasking];
} else {
    // ...
}
```

Network usage

In order to maintain connections in VoIP applications, iOS provides a mechanism to set a keepalive handler with setKeepAliveTimeout:handler: on UIApplication, which will be called automatically by the system. The minimum interval is 10 minutes, however, using slightly less than 30 minutes seems to be the most optimal for maximising battery life.

The operating system does not guarantee that keepalive handler will be called exactly at the requested time, as it performs various optimisations for waking up the system and aligning several timers to be triggered simultaneously.

To ensure optimal use of resources, apps should share a TCP connection where possible. Therefore, a single TCP session should be used for all communication but if this is not possible, no more than four TCP connections should be used at any one time.

Similarly, apps should not keep connections open when they have completed their task(s). All TCP sessions should be torn down correctly with FINs and should not be left in an undetermined state. This removes the need for a persisting state in the mobile (memory/battery) and in the network.

Device states

Apart from supporting multitasking, the app should also be aware of different states, such as screen lock/unlock, switching to phone call and back. In the main, this can be done by handling applicationDidBecomeActive:, applicationWillResignActive:. If there are any heavy operations that use the device's resources (graphics, network), then it would be better to suspend them if possible.

iOS also allows the app to prevent the device from going to sleep mode as follows

```
[[UIApplication sharedApplication] setIdleTimerDisabled: YES]
```

If the app relies on a network connection and needs to be connected even when the device is in sleep mode, then the parameter UIRequiresPersistentWi-Fi shouldl be added into Info.plist file. Without this parameter, any Wi-Fi connectivity will be disconnected after a while.

References Audio

Session in screen lock

https://developer.apple.com/library/ios/#documentation/Audio/Conceptual/AudioSessionProgrammingGuide/Introduction/Introduction.html

4.1.9 Scheduling

Scheduling the network activities of a third party app in synch with requests from other third party apps is not supported on iOS.

4.2 Android™

4.2.1 Asynchrony

Section 2.2.1 sets out the generic principles surrounding the use of asynchrony to enhance user experience during network activity. These principles are as applicable to Android as to other platforms.

It may be worthwhile extending the technique slightly to introduce some notion of "throttling" to avoid fully saturating the available bandwidth. This will improve the overall user experience (by allowing network requests from other parts of the UI to be met) and improve resilience to adverse network conditions (a fully saturated connection is more likely to lead to failed requests).

4.2.1.1 Implementation Details

Android implements the subset of the Apache Http APIs org.apache.http to support network activity using the "blocking" I/O model. Non-blocking I/O org.apache.http.impl.nio is not supported at present, so apps need to implement asynchrony using standard Java constructs and the supporting classes provided by the Android framework.

Figure 12 shows the classes required to support asynchrony in a simple Android app that fetches and displays a bitmap from the network in response to a button press. It consists of two activities MainActivity, ShowBitmapActivity and a helper class AsyncHttpReq.

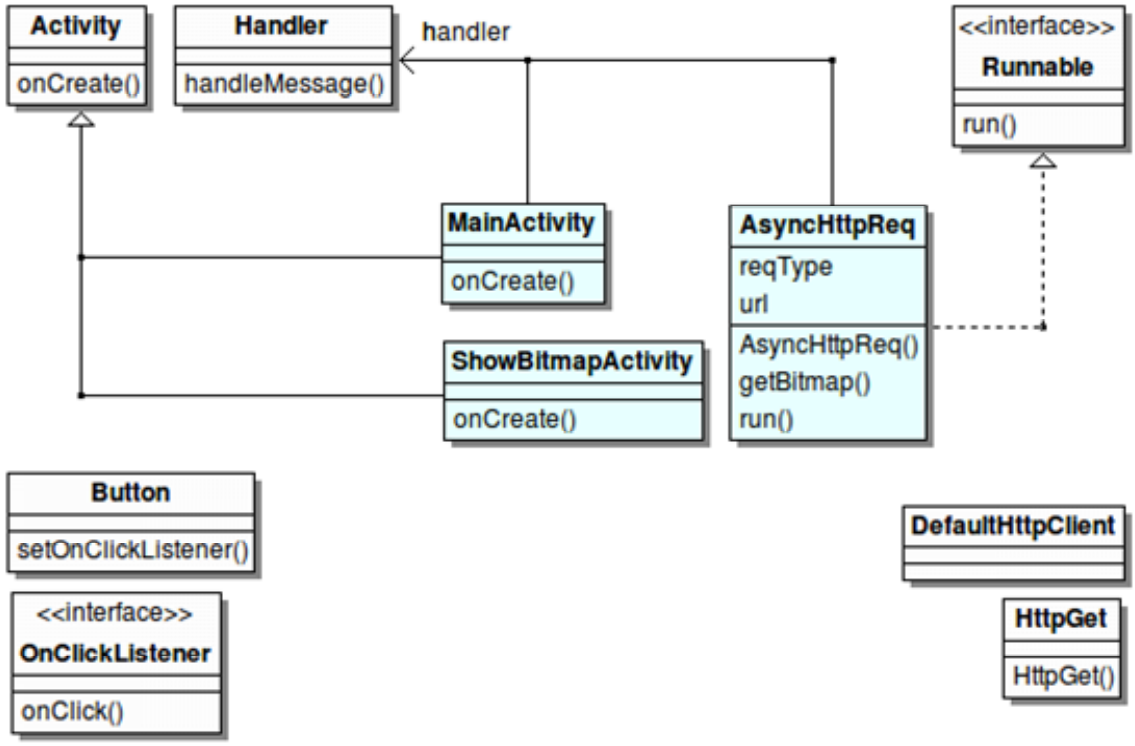

Figure 12: Asynchrony example

Figure 14 shows the execution sequence of the app. Broadly speaking this has five phases, each of which run asynchronously with each other.

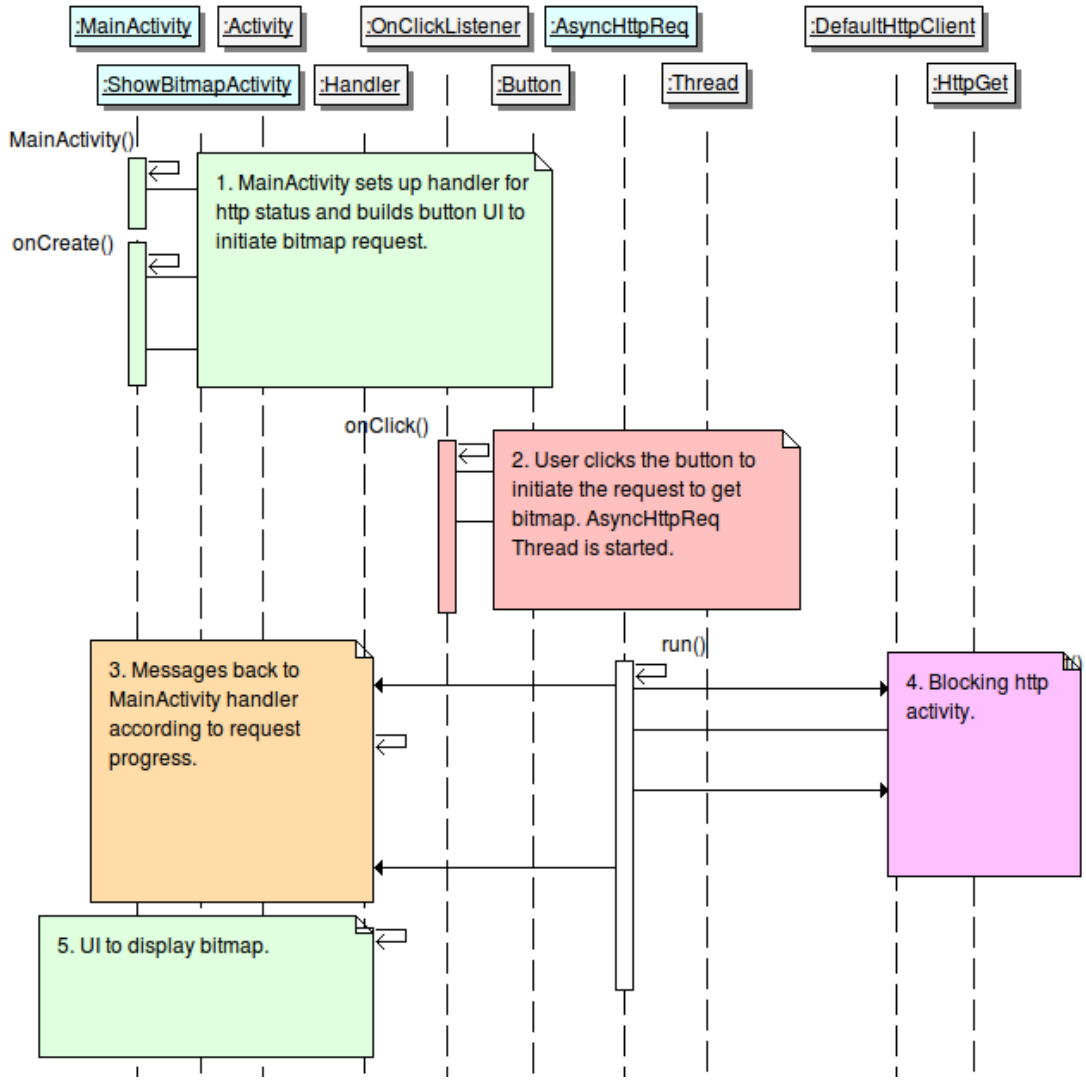

Figure 13: Asynchrony example – Sequence diagram

Each phase illustrates a different aspect of asynchrony:

- The application's MainActivity class is responsible for creating the initial UI (including a button click listener). More importantly it creates an instance of Handler that will be used to receive asynchronously coarse-grained status messages during the course of the HTTP request

The button listener's onClick method is invoked asynchronously when the user presses the button. At this point an instance of AsyncHttpReq is constructed and its Thread is started Some time later the Thread associated with the AsyncHttpReq will run. It sends a message back to the handler to indicate that processing of the request has begun. A further message is sent with the result when the request is complete (or times out). These messages are sent asynchronously

HTTP activity uses the blocking APIs and may take some time to complete depending on network conditions and image size

When the handler receives a message indicating that the request has been successful it starts the UI to display the bitmap

This example does not implement throttling, as each request will be processed as soon as the associated Thread is run, potentially saturating the available bandwidth. In its simplest form throttling could be implemented using a queue and limiting the number of simultaneously active requests.

After Android 1.5 a lightweight method is introduced by the platform to simplify the task. A utility class called AsyncTask is written to provide a simple way to achieve background processing, without worrying too much about the low-level details (threads, message loops etc). It provides call-back methods that help to schedule tasks and also to easily update the UI whenever required – demonstrated in the article “Painless Threading” (see below).

However, please bear in mind that AsyncTask is a lightweight solution with some limits:

- AsyncTask uses a static internal work queue with a hard-coded limit of 10 elements. Trying to download 30 images from the server, for example, would cause the work queue to quickly overflow and many tasks would get rejected
- AsyncTask can’t survive its Activity being torn down by the OS and recreated. If this behavior is not desired, it is recommended to use a Service instead
    a) AsyncTask is meant to be used for short operations of around a few seconds only
- It is not possible to interact with background thread and exceptions are not well handled

In most cases, AsyncTask is acceptable, but for complex cases where the above limits arise, you would need to build your own worker/handler solution.

References

Painless Threading

http://developer.android.com/resources/articles/painless-threading.html

4.2.1.2 Non-Blocking User Interface

Unresponsive user interfaces are perhaps the most common cause of user frustration with a particular app. With Android, unavoidable delays and unresponsive applications typically lead to a blocked UI.

Unavoidable Delays

It is quite common for an app to be faced with unavoidable delays, for instance when downloading large files from the network or processing large images. However even when a delay is unavoidable the app should try to ensure that the UI is not blocked.

Android provides some UI widgets to help to improve the user experience during these delays. When the delay is known to be brief (less than five seconds, for instance) it is acceptable to use the ProgressDialog (see Figure 14).

Figure 14: Android ProgressDialog

These still block the UI but are at least preferable to a frozen screen. For longer delays the ProgressBar view can be incorporated as part of the layout and driven from a helper thread associated with the current Activity. This allows the other parts of the UI to remain active.

Unresponsive Applications

When an app blocks the Android UI for more than a few seconds, the “Application Not Responding” (ANR) message will display, requiring the user to choose whether to continue with or abandon the app.

Figure 15: Activity Not Responding Dialog

Clearly this is highly undesirable. In most cases, the user will close the app, and the internal data-structures will need to be recomputed if the user subsequently reruns the app.

To avoid this, apps should be structured to minimise the amount of work done in any methods that run in the main thread (typically the Activity life-cycle methods e.g. onCreate(), onResume()). For Activities, the easiest way to do this is to offload the work to a child thread and provide a Handler class which the child can use to indicate when the work is complete.

Garbage collection can also lead to noticeable delays in the user interface. To a certain extent this is unavoidable but minimising the number of objects that are created in main thread methods will help.

4.2.2 Offline mode

A common problem with mobile app design is the default assumption that a connection is always available, and the lack of a connection is treated as a corner case or error condition. This approach is reinforced through the common use of emulators (which are effectively always connected) during the development process.

A safer approach is to assume that a connection is seldom available and design the architecture of the app accordingly. This approach tends to encourage the development of stronger abstractions between the app and its data and this in turn is likely to lead to an architecture that lends itself more easily to the kind of asynchronous implementation discussed earlier. Taking this approach to its logical extreme, apps would direct all network traffic to a local service implementing a shared intelligent local persistent cache.

Whatever approach you decide to take, Android provides several options for storing and accessing persistent data.

References

Data Storage

http://developer.android.com/guide/topics/data/data-storage.html

Bandwidth Awareness

The Android Connectivity Manager can be used to determine the connection state, and the application can also register to receive status updates. An instance of NetworkInfo can be obtained via the Android Connectivity Manager, and with it, it is possible to determine if the currently used Network interface is roaming and possibly limit network traffic.

Where a throttling strategy has been used, (as outlined in Section 4.2.1), the connection status can be used to dynamically alter the throttle settings e.g. to increase the number of simultaneous requests that are allowed when a Wi-Fi connection is present.

4.2.3 Caching

Android's browser makes use of internal APIs to support HTTP caching. These APIs are not available to other applications and so extra effort is needed if caching is to be supported. According to RFC-2616, a possible sequence diagram could be as in Figure 16 on the following page:

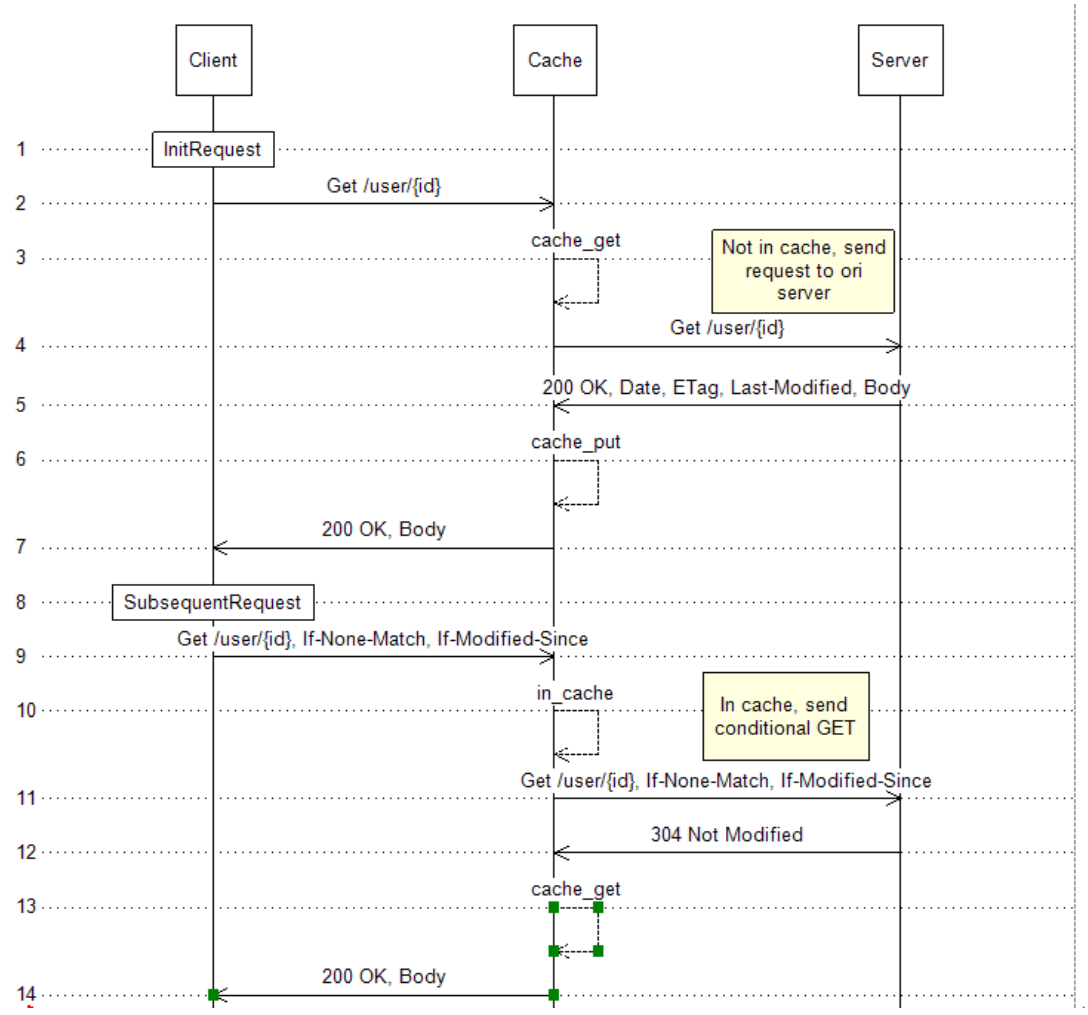

Figure 16: Http Caching – Sequence diagram

The full implementation of HTTP cache is laborious. Apache HttpClient implements CachingHttpClient, a drop-in replacement for a DefaultHttpClient, to provide an HTTP/1.1-compliant caching layer, but it is not available under Android. The flow in Figure 16 has to be implemented manually. A simple conditional get can be implemented as below:

```
class ConditionalGetExample {
    String entityTag = "";
    String lastModified = "";

    public void start() throws HttpException, IOException {
        HttpClient client = new DefaultHttpClient();
```

```
        HttpGet request = new HttpGet(("http://www.apache.org”);
        setHeaders(request);
        HttpResponse response = client.executeMethod(method);
        processResults(response);
    }



    private void setHeaders(HttpGet  request) {
        request.setHeader("If-None-Match", entityTag);
        request.setRequestHeader("If-Modified-Since", lastModified ));
    }

    private void processResults(HttpResponse response) throws HttpException {
        if(response.getStatusLine().getStatusCode() ==
            HttpStatus.SC_NOT_MODIFIED) {
            Log.d(“Http cache”,
                "Content not modified since last request");
            return;
        }
        else {
            entityTag = retrieveHeader(method, "ETag");
            lastModified = retrieveHeader(method, "Last-Modified");
            // process fresh content here!
        }
    }

    private String retrieveHeader(HttpResponse response, String name)
        throws HttpException {
        Header[] headers = response.getHeaders(name);
        String value = "";
        if(header.length > 0) {
            value = header[0].getName();
        }
        return value;
    }
}
```

If you are using HttpUrlConnection from the java.net package, then Java’s response cache mechanism might be another approach. There are three abstract classes: ResponseCache, CacheRequest, CacheResponse. You would need to extend these classes for your own cache implementation. The flow of events is something like the following:

- A concrete class of ResponseCache registers with the system by using the static method ResponseCache. setDefault(ResponseCache)

- There are two methods in the ResponseCache that are invoked by the protocol handlers. get() returns a CacheResponse and put() returns a CacheRequest
- The creation of a URLConnection and attempt to read content creates the appropriate stream handler, which checks for the content in the cache by invoking ResponseCache.get()
- If the content is found in the cache, it is returned. Otherwise a request is sent to the origin server, the response is sent to ResponseCache.put() to see if the content is cacheable (based on the response headers)

A reference implementation is in the article below. The work of cacheability determination, placing the resource content in the cache, evicting the content based on the “Expires” or “Date” headers, and retrieving the resource will be done by your own cache implementation.

References

Using ResponseCache in an Android App

http://codebycoffee.com/2010/06/29/using-responsecache-in-an-android-app/

4.2.4 Security

Android exposes a number of standard Java APIs to support security:

- java.security package provides classes and interfaces supporting the security framework:
  - Generation and storage of public cryptographic keys
  - Message digest and signature generation
  - Secure random number generation
- javax.crypto package provides additional classes and interfaces for common cryptographic operations:
  - Symmetric, asymmetric, block and stream ciphers
  - Secure streams and sealed objects
- javax.security.* packages
  - Authentication and authorisation
  - Public key certificates

The whole framework is “pluggable” in the sense that the underlying cryptography implementation is abstracted away from the public APIs so that 3rd party providers can be supported. Android makes use of two security providers: Bouncy Castle and the Apache Harmony APIs. These are explicitly instantiated in the `java.security` package, extensive use of which is made throughout other parts of the Android framework to support e.g. HTTPS, SSL Webkit browser etc.

Use of these APIs within the context of Android is not well documented. However as the APIs are standardized there are plenty of generic examples of their use.

4.2.5 Push notifications

Google Cloud Messaging (GCM) is the preferred method of pushing data to a device as it significantly impacts optimisation. Alternative methods such as SMS are also possible but not covered in this document.

- Existing applications using C2DM (Cloud To Device Messaging)are encouraged to migrate to GCM and any new application development should exclusively use GCM.
- GCM is only available for devices running Android version 2.2 and requires Google Play Store.
- As opposed to C2DM (Android’s old cloud messaging system), GCM does not impose any quota on the number of messages delivered to Android devices. GCM is not meant for pushing the entire payload to a device, but to notify the application that it can go fetch new information available in its server.

4.2.6 Data formats

Android includes support for both JSON and XML interchange formats. JSON support is provided by four classes in the org.json package:

- JSONArray – Indexed sequence of values
- JSONObject – Set of name/value mappings
- JSONStringer – String conversion
- JSONTokener – Parse JSON encoded strings into corresponding objects
- android.util – Provides two further classes for reading/writing JSON encoded stream of tokens (includes examples)

JSON is standardised through RFC 4627 and so not unsurprisingly plenty of examples of its use can be found on the web. The Android framework itself makes fairly extensive use of JSON – for example Android In-App Billing transaction information is contained in a JSON string.

Android support for XML is provided by the following packages:

- org.xml.sax – Core SAX APIs ¦ org.xmlpull – Support for XML pull parsing
- javax.xml.datatype – XML/Java type mappings
- javax.xml.namespace – XML namespace processing
- javax.xml.parsers – Processing for XML documents supporting pluggable parses for SAX and DOM
- javax.xml.transform – Transformations from Source to Result with support for DOM, SAX2 and stream- and URI-specific transformations
- javax.xml.validation – API for the validation of XML documents
- javax.xml.xpath – API for the evaluation of XPath expressions (a simple concise language for selecting nodes from an XML document)
- android.xml – XML utility methods

Again the Android framework makes extensive use of XML – e.g. the Android application manifest, UI layout files and internationalisation all use XML.

References

http://developer.android.com/reference/org/json/package-summary.html

4.2.7 Compression

Android supports gzip and deflate compression for HTTP content. However compression is not enabled by default and so developers need to explicitly add the “Accept-Encoding” header to the request and handle the received content according to its “Content-Encoding” header.

The following sample code shows how to add compression using interceptors with the Apache HTTPClient:

```
DefaultHttpClient httpclient = new DefaultHttpClient();

// Add gzip header to requests using an interceptor

httpclient.addRequestInterceptor(new GzipHttpRequestInterceptor());


// Add gzip compression to responses using an interceptor

httpclient.addResponseInterceptor(new GzipHttpResponseInterceptor());


...
```

```
//Request interceptor for adding gzip accept header
private final class GzipHttpRequestInterceptor implements HttpRequestInterceptor {
       public void process(final HttpRequest request, final HttpContext context) throws HttpException, IOException {
           if (!request.containsHeader("Accept-Encoding")) {
               request.addHeader("Accept-Encoding", "gzip");
           }
       }
}

//Response interceptor for handling compressed responses
private final class GzipHttpResponseInterceptor implements HttpResponseInterceptor {
       public void process(final HttpResponse response, final HttpContext context) throws HttpException, IOException {
           HttpEntity entity = response.getEntity();
           Header header = entity.getContentEncoding();
           if (header != null) {
               HeaderElement[] codecs = header.getElements();
               for (int i = 0; i < codecs.length; i++) {
                   if (codecs[i].getName().equalsIgnoreCase("gzip")) {
                       response.setEntity(new GzipDecompressingEntity(response.getEntity()));
                       return;
                   }
               }
           }
       }
}

//Compression entity used in response interceptor
static class GzipDecompressingEntity extends HttpEntityWrapper {
       public GzipDecompressingEntity(final HttpEntity entity) {
            super(entity);
        }

        @Override
        public InputStream getContent() throws IOException, IllegalStateException {
            InputStream wrappedin = wrappedEntity.getContent();
            return new GZIPInputStream(wrappedin);
        }

        @Override
        public long getContentLength() {
```

```
            return -1;
        }
}
```

4.2.8 Background / Foreground modes

Android recognises five different process states:

- Foreground process: This is the part of the app that is currently visible and with which the user is interacting. More precisely an Activity is considered to be in the foreground between calls to the onResume() and onPause() methods, so typically the onPause() method is where application data ought to be persisted and CPU intensive tasks terminated (e.g. application threads, animation + other content rendering)
- Visible process: Although this part of the app is no longer in the foreground some of its UI components are visible. An example of this would be when a dialog box is in the foreground partially obscuring the other activity's UI
- Service process: These are started by startService() and do not fall into either of the previous process states as services do not present a UI. Service processes keep running until they are explicitly stopped or the system runs out of memory
- Background process: Activities whose onStop() method has been called. As no part of the activity's UI is visible it is assumed that these processes can be killed at any time (by implication this means that the application must have saved its state correctly as a side effect of earlier life-cycle methods)
- Empty process: These are retained to improve start-up performance of other components

For Activities Android also provides the onSaveInstanceState() to help with persistence. This is called (immediately prior to onPause()) when the app is to be destroyed by the system . This allows discrimination between this condition (when the user might expect the app's last state to be restored the next time it is run) and when the user has shut down the app (and might therefore expect it to run from the start next time).

For activity lifecycle methods from onPause() onwards the application process can be killed at any time after the method returns.

Services should not in general run in the foreground, nor should they run continuously in the background. They should instead be triggered by some event or woken up periodically to perform a task, and then call stopService(), in order to minimise memory and CPU consumption and to avoid the risk of being killed by the OS.

As mentioned in Section 3.6, if developers want their apps to obtain/update the latest information at the time when users start interacting with a device already in sleep mode, it is recommended to trigger the related network activity when the device screen is unlocked (rather than when the screen display just turns on). In terms of Android, this means using the ACTION_USER_PRESENT intent to trigger such network activity, rather than using the ACTION_USER_SCREEN intent. The ACTION_USER_PRESENT intent is notified when the device screen is unlocked, whereas the ACTION_USER_SCREEN intent is also notified when the screen display is just turned on.

4.2.9 Scheduling

Scheduling an app's network activities in synch with network activity requests from other apps will reduce the signalling load as they are batched together. This is achieved by using AlarmManager with setInexactRepeat and an interval constant provided in AlarmManager, e.g. INTERVAL_HALF_HOUR. These constants are special in that AlarmManager. setInexactRepeat will fire the intents at a regular intervals simultaneously, but not at exactly specified times

```
AlarmManager am = (AlarmManager)context.getSystemService(Context.ALARM_SERVICE);
Intent intent = new Intent(context, Poller.class);
PendingIntent pIntent = PendingIntent.getService(context,0,intent, 0);
long interval = AlarmManager.INTERVAL_FIFTEEN_MINUTES;
long firstPoll = System.currentTimeMillis();
am.setInexactRepeating(AlarmManager.RTC, firstPoll, interval, pIntent);
```

The alarm type can be either AlarmManager.RTC, which uses absolute time (wall clock time), or AlarmManager.ELAPSED_REALTIME, which uses relative time (time since boot).

Unless there really is a need to wake up the application at the scheduled alarm time, ensure that AlarmManager.RTC / AlarmManager.ELAPSED_REALTIME is used and not AlarmManager.RTC_WAKEUP/AlarmManager.ELAPSED_REALTIME_WAKEUP, i.e. don't wake up the application unless some other app is also being woken up. On devices with Google services, schedulers run regularly and will make sure the specified app wakes up now and then.

4.2.10 Spreading network activity timing among different devices

As mentioned in Section 3.7, it is recommended to design apps to spread network activity timing across different devices as much as possible. Some tips in designing Android apps to fulfil this goal are explained below.

Apps requiring periodic network activity but not necessarily at exact times

For applications requiring periodic network activities but not necessarily at exact times (e.g. the RSS newsfeed example mentioned in Section 3.7), it is ideal to evenly spread the network activity timings across devices.

One way to realise such behaviour would be to:

- Use *setRepeating* / *setInexactRepeating* of AlarmManager to schedule alarms for an app with:
  Alarm type set to AlarmManager.ELAPSED_REALTIME; and
  The base timing of the first alarm set to a timing which would be spread across devices (e.g. the timing when the app's activity is displayed).

Apps requiring network activity at exact times

For applications requiring network activities at exact times of a day (e.g. the weather widget example mentioned in Section 3.7), it is better to spread the network activity timings across devices within an acceptable time window.

One way to realise such behaviour would be to:

- Use *set* / *setRepeating* / *setInexactRepeating* of AlarmManager to schedule alarms for an app with:
  Alarm type set to AlarmManager.RTC; and
  Timing of the (first) alarm set to a summed value of:
  The desired exact timing (e.g. 17hr:00min); and
  A random offset within the acceptable time window (e.g. a random offset obtained using a uniform distribution function between 0min and 5min).

Synched NW activity timing of an app due to WAKEUP of another app

By using the mechanisms mentioned above, it is possible to spread network activity timings of an app across devices. However, even for apps designed with such mechanisms, the network activity timings may be synched across devices due another app with alarms scheduled at exact timings using the WAKEUP functionality (i.e. alarm type set to AlarmManager.RTC_WAKEUP). This behaviour is illustrated in Figure 16a below.

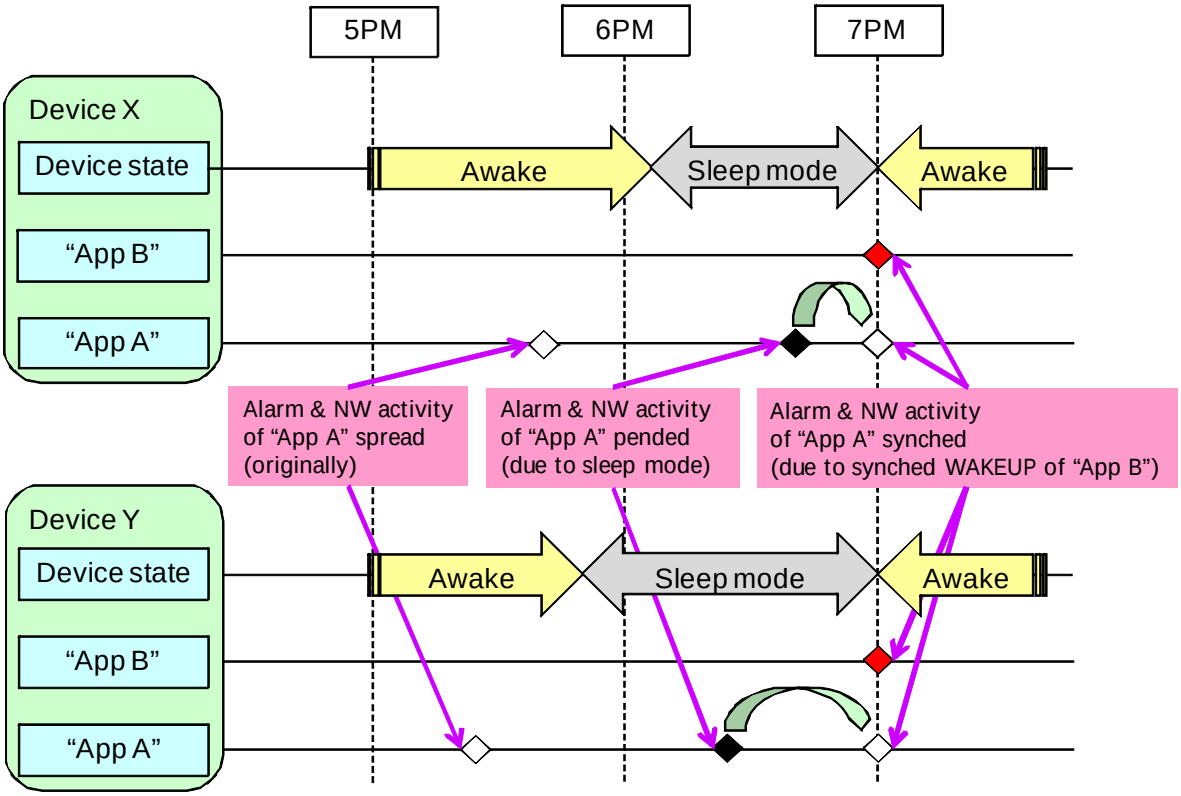

Figure 16a: Synched NW activity timing due to WAKEUP of another app

In Figure CC, "App A" and "App B" are both installed on Device X and Device Y, where:

- "App A":
  Uses repeated alarms with a 1 hour interval for periodic network activities;
  Does not use the WAKEUP functionality;
- "App B":
  Has an alarm set at 7PM on both Device X and Device Y;
  Uses the WAKEUP functionality.

Originally, the alarm / network activity timing of "App A" is designed to be spread across Device X and Device Y (i.e. up to the 5PM hour). By the alarm timing of "App A" in the 6PM hour, Device X and Device Y have gone to sleep mode, and hence the intents / network activity for "App A" are pended in both devices. At 7PM, Device X and Device Y wake up due to the scheduled alarm of "App B" using the WAKEUP functionality. The pended intent of "App A" is fired at this timing, and results in synched network activity of Device X and Device Y. Note that "App B" may be an alarm clock, which does not incur network activity by itself.

One way to avoid such unintended synching of an app's network activity timing across devices would be to:

- Utilize the sleep method of Thread class (i.e. Thread.sleep) to suspend an app's task (even for a short period of time), where:
- When receiving intents from the AlarmManager for an app, Thread.sleep (with even a short period of sleep time) is called before actually running the tasks for that app; and
- A random sleep time (e.g. a random time obtained using a uniform distribution function between 0min and 1min) is used.

With such careful implementation, for the example illustrated in Figure 16a, the network activity timing of "App A" at 7PM (due to the wake up caused by "App B") can be spread to a certain extent across Device X and Device Y.

It is noted that periodic network activity of an app can also be realised by using AlarmManager with set (instead of setRepeating / setInexactRepeating), i.e. by entering set after every time tasks for that app are invoked. However, such design will increase the probability of synched network activity timing of the app across devices. For example, in the case illustrated in Figure 16a, the network activity timing for “App A” is synched across Device X and Device Y at the 7PM due to the WAKEUP event caused by “App B”. If “App A” has used set to schedule the hourly alarms, the periodic alarm / network activity timing for “App A” from thereon will be continuously synched across Device X and Device Y. By using setRepeating or setInexactRepeating instead, such unintended periodic synching can be avoided, since the base time for the periodically repeated alarms are kept unchanged. Therefore, it is recommended to use setRepeating / setInexactRepeating when scheduling alarms periodically.

Alarm type for setInexactRepeating

As mentioned above, using AlarmManager with setInexactRepeating is useful in bundling network activities of multiple apps within a device and spreading network activity timings across different devices. However, for earlier Android OS versions (e.g. Gingerbread and earlier), it has been noticed that when AlarmManager.RTC is used for the alarm type, the alarms are scheduled at specific wall clock times (e.g. XXhr:00min, XXhr:15min, XXhr:30min, XXhr:45min). This may result in unintended concentration of network activities across different devices, and hence the use of AlarmManager.ELAPSED_REALTIME rather than AlarmManager.RTC is recommended.

4.3 Windows Phone

For a general overview of Windows Phone networking, please refer to the following article:

Windows® Networking in Silverlight for Windows Phone

http://msdn.microsoft.com/en-us/library/ff637320(VS.95).aspx

4.3.1 Asynchrony

Windows Phone 7 is the newest smartphone platform. It is designed to operate in multithreaded mode and support asynchronous communication out of the box. All network access in WP7 is asynchronous and the main networks APIs do not expose synchronous methods to minimise the impact on the performance of the UI. As network resources can be accessed in a number of ways, it is important to understand the multithreaded architecture.

Every Silverlight application will have the following threads by default:

- **UI thread** – responsible for handling user input, drawing new visuals, and calling back to user code
- **Main thread** – responsible for handling user code, such as loading and processing of data, implementing business logic, etc.

It is essential to keep the UI thread as free as possible, as maintaining a lightweight UI thread is the key factor of a responsive app. Access to the network resources can be performed from both the user code and XAML mark-up. All objects referenced from XAML are downloaded and processed asynchronously by the Silverlight engine. Network resources accessed from the user code are handled by the APIs of the System.Net namespace which includes:

- WebClient class – provides common methods for sending data to and receiving data from a resource identified by a URI.
- WebClient is a wrapper class around the HttpWebRequest class and can be easier to use because it returns result data to the app on the UI thread. WebClient supports

events. WebClient is a higher level API than HttpWebRequest with callbacks made on UI thread and support of events

- HttpWebRequest class – is a lower level API compared with WebClient and provides richer functionality and better control over HTTP communication. Callbacks are implemented through a delegate function and made on the Main thread

Although both of these classes support asynchronous communication only, it is important to understand the differences between them.

WebClient class

This class is designed for use from Silverlight controls that are hosted in a XAML page. It provides the simplest way of accessing network resources but must be used carefully as it operates in the UI thread and canimpact UI responsiveness. Developers should ensure that all code referenced by event handlers only performs tasks that are related to updating the UI, otherwise this will delay the return of control to the UI thread and make UI operation sluggish.

Here is an example:

```
try
{
    System.Uri uri = new Uri("http://www.bing.com");
    WebClient webClient = new WebClient();

    // Assign callback event handler
    webClient.OpenReadCompleted +=
        new OpenReadCompletedEventHandler(webClient_OpenReadCompleted);
    // Create a HttpWebrequest object to the desired URL
    webClient.OpenReadAsync(uri);
}
catch (Exception ex)
{
    // TODO: your exception handling code
    webClientTextBlock.Text =
        "Exception raised! Message: " + ex.Message;
}



void webClient_OpenReadCompleted(object sender, OpenReadCompletedEventArgs e)
{
    try
    {
        using (StreamReader reader = new StreamReader(e.Result))
        {
            webClientTextBlock.Text = reader.ReadToEnd();
        }
    }
```

```
    catch (WebException ex)
    {
        // TODO: your exception handling code
        webClientTextBlock.Text =
            "WebException raised! Message: " + ex.Message +
            "\nStatus: " + ex.Status;
    }
    catch (Exception ex)
    {
        // TODO: your exception handling code
        webClientTextBlock.Text =
            "Exception raised! Message: " + ex.Message;
    }
}
```

HttpWebRequest class

This class is designed for use from the user code and should normally be considered for accessing data feeds, submitting data to the cloud and manual handling of static network resources. Callbacks are processed via AsyncCallback delegate function and are made on the Main thread. This means that all code updating the UI must be synchronised with the UI thread, otherwise access to any UI controls or UI related classes (such as BitmapImage) will result in a System. InvalidOperationException. In order to synchronise output with the UI thread it is necessary to invoke the code through a dispatcher.

```
Dispatcher.BeginInvoke(() => { /* UI update code */ });
```

Before invoking the dispatcher, you should ensure the processing of all non-UI related data is complete. Keep this block of code as compact as possible, only perform UI updates and don't perform any unnecessary calculations as anything executed here may further delay the rendering of the UI.

Here is an example of how to use the HttpWebRequest class:

```
try
{
    System.Uri uri = new Uri("http://www.bing.com");
    // Create a HttpWebRequest object to the desired URL
    HttpWebRequest httpWebRequest = (HttpWebRequest)WebRequest.Create(uri);

    // Start the asynchronous request
    IAsyncResult result = (IAsyncResult)httpWebRequest.BeginGetResponse(
        new AsyncCallback(ResponseCallback), httpWebRequest);
}
catch (WebException ex)
```

```
{
    // TODO: your exception handling code
    Dispatcher.BeginInvoke(() =>
    {
        httpWebRequestTextBlock.Text =
            "WebException raised! Message: " + ex.Message +
            "\nStatus: " + ex.Status;
    });
}

catch (Exception ex)
{
    // TODO: your exception handling code
    Dispatcher.BeginInvoke(() =>
    {
        httpWebRequestTextBlock.Text =
            "Exception raised! Message: " + ex.Message;
    });
}



private void ResponseCallback(IAsyncResult result)
{
    try
    {
        HttpWebRequest httpWebRequest = (HttpWebRequest)result.AsyncState;
        // Obtain WebResponse from the callback result parameter
        WebResponse webResponse = httpWebRequest.EndGetResponse(result);

        using (Stream responseStream = webResponse.GetResponseStream())
        using (StreamReader responseStreamReader =
            new StreamReader(responseStream))
        {
            // Read response body
            string contents = responseStreamReader.ReadToEnd();
            // Invoke dispatcher to access UI thread
            Dispatcher.BeginInvoke(() =>
            {
                // Update UI control
                httpWebRequestTextBlock.Text = contents;
            });
        }
```

```
    }
    catch (WebException ex)
    {
        // TODO: your exception handling code
        Dispatcher.BeginInvoke(() =>
        {
            httpWebRequestTextBlock.Text =
                "WebException raised! Message: " + ex.Message +
                "\nStatus: " + ex.Status;
        });
    }
    catch (Exception ex)
    {
        // TODO: your exception handling code
        Dispatcher.BeginInvoke(() =>
        {
            httpWebRequestTextBlock.Text =
            "Exception raised! Message: " + ex.Message;
        });
    }
}
```

Managing asynchronous requests

In most cases the monitoring of asynchronous requests is not necessary as responses will automatically fail in the case of network errors resulting in WebException being raised. Network error exceptions must be always handled appropriately by code, but in some scenarios it will be necessary to implement the following:

- **Timeout handling** – in asynchronous programming, it is the responsibility of the client application to implement its own time-out mechanism
- **Cancellation of requests** – for example when the user wants to manually terminate network requests

If an app requires management of asynchronous requests for any reason, references to all issued requests need to be stored and their states passed across asynchronous calls within the thread. This technique is based on storing parameters related to individual requests in the RequestState class. To implement timeout handling and cancellation of asynchronous requests, follow this article:

http://msdn.microsoft.com/en-us/library/system.net.httpwebrequest.abort(v=vs.80).aspx

***Note**: Always remember that WebClient callbacks are made on the UI thread*

***Note:** Keep non-UI related code out of the dispatched code when using HttpWebRequest*

References

HttpWebRequest class

http://msdn.microsoft.com/en-us/library/system.net.httpwebrequest(v=VS.95).aspx

WebClient class

http://msdn.microsoft.com/en-us/library/system.net.webclient(v=VS.95).aspx

Understanding Threads

http://msdn.microsoft.com/en-us/library/ff967560(v=vs.92).aspx#BKMK_Threads

Making Asynchronous Requests

http://msdn.microsoft.com/en-us/library/86wf6409.aspx

4.3.2 Connection loss and error handling

Regardless of the network error recovery strategy chosen for an app, it is absolutely necessary to ensure that:

- Your app never crashes due to a network error
- Your app informs the user about network issues in an unobtrusive way

The application error recovery strategy may be as simple as instructing the user to restart the app in order to refresh, or may be more sophisticated implementing offline mode with manual or automatic retry or network status monitoring features. Depending on the chosen strategy, this list of facts and options is worth considering:

XAML referenced network resources are loaded automatically by Silverlight and are difficult to monitor. You will have to load these resources into the UI yourself in order to have a better control

- Always catch exceptions from `WebCllient` or `HttpWebRequest` when initiating requests and reading responses, otherwise your application will crash. The `WebException.Status` property contains a `WebExceptionStatus` value that indicates the source of the error. Code samples for your convenience are given in the previous section **Error! Reference source not found. Error! Reference source not found.**.
- Monitor errors globally and integrate network health flags into ViewModel so that you can consistently report connectivity errors and perhaps offer a recovery action

If all network resources are managed through user code, then consider the following:

- Design the data transfer routine so it can be restarted at any time. Different parts of an app will likely have different routines
- Manage asynchronous network requests to allow safe cancellation of them. Example technique is covered in the previous section 4.3.1 Asynchrony -> Managing asynchronous requests
- Store successfully loaded resources in persistent storage, so they are not reloaded every time – this will be a part of an offline mode implementation. Don't forget to clean up the cache. See sections 4.3.3 Caching for extra information
- Monitor connection status events in order to automatically restart failed data transfer routines

Implementation of good recovery strategy is simpler if the app follows the Model-View-ViewModel (MVVM) design pattern. Errors are dealt with at the data layer and status reported to the UI through ViewModel.

MVVM design pattern is well covered in this article: http://msdn.microsoft.com/en-us/magazine/dd419663.aspx

Automatic retry

Consider implementing automatic retry only when performing lengthy or scheduled data transfers. Don't perform the retry immediately after the failure, as the failed network interface requires time to recover. Also, limit the number of retries, otherwise the app can drain the device battery. Here is an example of a good algorithm:

- First retry after one minute
- Second retry after five minutes
- Third (and last) retry after 15 minutes

Tip: Consider Automatic retry for network applications which work under Lock Screen

Checking network connection status

Checking connection status on Windows Phone 7 is impractical as this information is not immediately available to the app. It may take a couple of seconds to determine the type of connection.

This information is obtained via the NetworkInterfaceType property of the NetworkInterface class of the Microsoft.Phone. Net.NetworkInformation namespace. The NetworkInterfaceType property returns one of the following values:

- Wireless80211
- Ethernet
- MobileBroadbandGSM
- MobileBroadbandCDMA
- None

Below is an example of how to obtain status and monitor changes of network connection type:

```
using System;
using System.Net.NetworkInformation;
using System.Threading;
using System.Windows;
using Microsoft.Phone.Controls;
using Microsoft.Phone.Net.NetworkInformation;

namespace ConnectionStatus
{
    public partial class MainPage : PhoneApplicationPage
    {
        // Indicates type of the current connection to the internet
        private NetworkInterfaceType internetConnectionType;

        // Main Page Constructor
        public MainPage()
        {
            InitializeComponent();

            // Subscribes to the Network Address Change notifications
            NetworkChange.NetworkAddressChanged += new
```

```
        NetworkAddressChangedEventHandler(NetworkChange_NetworkAddressChanged);
        }

        // Standard Page_Loaded event handler
        private void PhoneApplicationPage_Loaded(object sender,
            RoutedEventArgs e)
        {
            CheckCurrentNetworkType();
        }

        // Network Address Changed notifications event handler
        private void NetworkChange_NetworkAddressChanged(object sender,
            EventArgs e)
        {
            CheckCurrentNetworkType();
        }

        private void CheckCurrentNetworkType()
        {
            // Checking the network type is not instantaneous
            // so it is advised to always do it on a background thread
            ThreadPool.QueueUserWorkItem((o) =>
            {
                // Determining type of current network interface
                internetConnectionType =
                    Microsoft.Phone.Net.NetworkInformation.
                    NetworkInterface.NetworkInterfaceType;

                // Synchronizing with the UI thread in order to update control
                Dispatcher.BeginInvoke(() =>
                {
                    textBlockConnectionType.Text =
                        internetConnectionType.ToString();
                });
            });
        }
    }
}
```

Bandwidth awareness

In general, app developers do not care how the app's interaction with the Internet is routed; i.e. whether it goes over a mobile or Wi-Fi connection. However for some apps the type of connection matters, such as those that offer an enhanced experience over a high-bandwidth Wi-Fi connection or those that aim for efficient use of a broadband mobile network.

References

NetworkInterface class

http://msdn.microsoft.com/en-us/library/microsoft.phone.net.networkinformation.networkinterface.networkinterfacetype%28v=VS.92%29.aspx

4.3.3 Caching

Caching of network resources is only partially supported by the Silverlight engine. The Silverlight UI has in-memory resource cache designed for improving rendering performance and optimisation of memory. In most cases this cache is filled with local resources, but network resources can be referenced as well. In general this only provides an advantage for network resources referenced multiple times from within the XAML mark-up. As this cache is stored in memory it isn't persistent and all data is lost with every process restart. When it is most important to improve the performance of apps during the tombstoning cycle, this feature becomes absolutely impractical as all data loaded from the network is lost as soon as the app is closed or suspended.

The bad news with respect to HTTP caching is that neither WebClient nor HttpWebRequest implement any caching features. Additional effort is required by the developer, but depending on requirements it might be possible to put in place some simple workarounds:

- **Cached data feed** – before parsing a recently downloaded XML or JSON data feed, first save it locally and then parse. Next time check whether the data has to bereloaded based on a fixed time interval. If not, load it from the local storage and process. This provides a good solution to Windows Phone Tombstoning
- **Image Cache** – the approach here is to save downloaded images to the isolated storage first and then update the UI. If the namespace of image URL references is consistent it may be possible to implement simple naming for images files stored in one folder; otherwise a more sophisticated naming algorithm is needed. For any subsequent requests, first check whether the app already has a copy of the requested image, and then load it from the folder. This can be good for images that never change, otherwise a content expiration policy needs to be implemented and integrated with the server

A few other tips:

- Bear in mind that not all data should be cached - it depends on the sensitivity of the information, its dynamic nature, etc.
- Some data may never change or expire, such as logos
- Design independent data loader classes which can be reused throughout the app
- Don't integrate the data loader with the ui, always feed data through view model
- Use httpwebrequest class with data loader, as it enables access to storage apis and viewmodel on the main thread from the callback delegate; synchronisation with the ui thread will happen in viewmodel
- The server component may already implement an expiration policy and communicate it via cache-control, last-modified or etag http headers (see rfc 2616 for full specification). It is relatively easy to read these attributes from webresponse class

Below are a few examples from the Windows Phone 7 community:

Offline Data Cache in Windows Phone 7

http://blogs.msdn.com/b/ukadc/archive/2010/10/21/offline-data-cache-in-windows-phone-7.aspx

Image Caching on Tombstoning

http://briankassay.com/blog/?p=95

4.3.4 Security

Secure HTTPS communication is transparently supported by Windows Phone at all levels including Silverlight UI framework (XAML referenced resources) WebClient and HttpWebRequest APIs:

```
// Uri to secure resource
System.Uri uri = new Uri("https://service.live.com");
WebClient webClient = new WebClient();
```

Or

```
// Uri to secure resource
System.Uri uri = new Uri("https://service.live.com");
// Create a HttpWebRequest object to the desired URL
HttpWebRequest httpWebRequest = (HttpWebRequest)WebRequest.Create(uri);
```

Authentication

Windows Phone only supports Basic Authentication protocol which does not encrypt user name or password. However, in order to implement secure authentication using Basic Authentication protocol, it is only necessary to ensure that:

- Communication at the time of authentication is performed over encrypted HTTPS connection
- Data exchange after authentication is always performed over HTTPS connection

It is not enough to exchange encrypted user credentials over unencrypted HTTP protocol or authenticate the user over HTTPS and then communicate over HTTP. The user's Authentication token can be stolen and credentials compromised.

Tip: Always communicate over HTTPS when using unencrypted authentication otherwise user credentials will be compromised

Mutual authentication

Although trusted certificates can be installed on the Windows Phone, in the current release, the platform does provide access to the installed certificates from apps. As a result mutual authentication scenarios – when the client sends its own certificates to the web service in addition to receiving one – cannot be implemented.

Storing user credentials

Isolated storage on Windows Phone is considered secure. However, you should make additional efforts to encrypt user credentials before saving. This will protect the information if access to the application storage is obtained through physical device theft, following a jailbreak, or via accessing devices' backup files. Windows Phone supports a number of encryption protocols including AES, SHA1 and SHA256.

Use an example from the following source in order to encrypt your data:

http://robtiffany.com/windows-phone-7/dont-forget-to-encrypt-your-windows-phone-7-data

References

Web Service Security for Windows Phone

http://msdn.microsoft.com/en-us/library/gg521147(v=vs.92).aspx

Security for Windows Phone

http://msdn.microsoft.com/en-us/library/ff402533(v=VS.92).aspx

System.Security.Cryptography Namespace

http://msdn.microsoft.com/en-us/library/system.security.cryptography(v=VS.95).aspx

4.3.5 Push notifications

Microsoft Push Notification Service (MPNS) provides a resilient, dedicated, and persistent channel to send data to a Windows Phone application from a web service in a power-efficient way. Each device maintains one connection with MPNS to receive notifications from the cloud. MPNS prioritises delivery of notifications and differentiates between Immediate, Medium and Low priorities. Medium and Low priority notifications are normally delayed and aggregated with other messages to reduce impact on device battery life, network traffic and therefore network signalling. MPNS also differentiates between Toast, Tile and RAW Notifications.

More information on APNs is available at:

http://msdn.microsoft.com/en-us/library/ff402537(v=vs.92).aspx

4.3.6 Data formats

Windows Phone 7 provides extensive support for XML-based services out-of-the-box including WCF, XML serialisation, DOM parser, Language Integrated Query (LINQ), XSD validation, etc.

JSON is only partially supported, with a limited serialiser. For richer support of JSON, consider Open Source product Json. NET (see reference below) which has a more flexible serialiser, LINQ, conversion of JSON to and from XML.

References

Json.Net

http://json.codeplex.com/

DataContractJsonSerializer class

http://msdn.microsoft.com/en-us/library/system.runtime.serialization.json.datacontractjsonserializer(v=VS.95).aspx

4.3.7 Compression

The current Windows Phone 7 APIs do not support HTTP compression and fixing this issue is extremely difficult due to a number of limitations. Although decompressing content encoded with GZip and Deflate algorithms is not difficult, and open source libraries such as Silverlight SharpZipLib are available, integration with WebClient and HttpWebRequest APIs is currently not possible due to protection put on HTTP Header “Accept-Encoding”.

Example:

```
// The following code will throw ArgumentException with the following message:
// The 'Accept-Encoding' header cannot be modified directly.
// Parameter name: name
httpWebRequest.Headers[HttpRequestHeader.AcceptEncoding] = "gzip";
```

4.3.8 Background / Foreground modes

Third party apps on the Windows Phone platform can only work in the foreground and under the Lock Screen if permitted to do so. Every time a normal app loses focus or calls Launcher or Chooser, it is immediately instructed to shut down and is given 10 seconds to save data. This process, called Tombstoning, supports the navigation interface and app with some tools allowing the app to restore, save and then restore its state when reactivated. Tombstoning, if not properly addressed, can have a significant impact on network traffic, and user experience in general as apps often reload data from the network when reactivated. This is considered bad practice and should always be addressed by developers. Where appropriate, delay frequent updates – for instance don't refresh a weather feed if the data was loaded minutes ago.

References

Silverlight SharpZipLib

http://slsharpziplib.codeplex.com/

Execution Model for Windows Phone (Tombstoning)

http://msdn.microsoft.com/en-us/library/ff769557(v=vs.92).aspx

4.3.9 Scheduling

Although this document references features and SDK of Windows Phone's first release 7.0, the major release of Windows Phone 7.5 'Mango' addresses the need for scheduling of some background activities via Background Agents. These allow an app to execute code in the background, even when the app is not running in the foreground. Different types of Scheduled Tasks are designed for different types of background processing scenarios, with different behaviours and constraints.

- PeriodicTask – Implements periodic agent which runs for a small amount of time on a regular recurring interval
- ResourceIntensiveTask – Implements periodic agent which runs for a relatively long period of time when the phone meets a set of requirements

Each app may have only one background agent, implemented as an app component but with its lifecycle managed independently by the OS. In a simplistic example, the OS would periodically execute all registered agents in turn, so that only one agent is active at a time and has a limited time to run. As soon as one agent finishes its procedure or gets terminated because of reaching duration limit, the next agent is executed, and so on until the cycle is complete. PeriodicTask and ResourceIntensiveTask agents have different schedule, duration and constraints, in order to optimise power consumption, network signalling and traffic, and minimise the impact on the user experience.

For an overview, best practices and implementation details, refer to: http://msdn.microsoft.com/en-us/library/hh202961(v=vs.92).aspx

Windows Phone SDK 7.1 includes a project template called Windows Phone Scheduled Task Agent for use when implementing a background agent.

5 References

- AQUA (App Quality Alliance) 'Best Practice Guidelines, AQuA Test criteria for Android'
- Certificate, Key, and Trust Services Programming Guide, Apple, 2010
- Cocoa Fundamentals Guide, Apple
- Error Handling Programming Guide, Apple
- Keychain Services Programming Guide, Apple, 2010
- Network Efficiency Task Force Fast Dormancy Best Practices; GSM Association
- RFC2616 – Hypertext Transfer Protocol HTTP/1.1
- RFC2617 – HTTP Authentication: Basic and Digest Access Authentication
- Security Overview, Apple, 2010
- Secure Coding Guide, Apple, 2010
- URL Loading System Programming Guide, Apple
- WWDC 2010 Sessions 105, 109 - Adopting Multitasking on iPhone OS
- WWDC 2010 Session 200 - Core OS Networking
- WWDC 2010 Sessions 207, 208 - Network Apps for iPhone OS

Other Information

It is our intention to provide a quality document. If you find any errors or omissions, please contact us with your comments. You may notify us at devguide@gsm.org

Your comments, suggestions and questions are always welcome.

Acknowledgements

Document Management

Document History

Version	Date	Brief Description of Change	Approval Authority	Editor / Company
1.0	February 2012	Initial published version providing a guide to improve apps connectivity, power consumption, user experience, security, and device battery life. http://www.gsma.com/technicalprojects/smarter-applications	TSG, PSMC	Kamran Kordi (Deutsche Telekom AG)
2.0	February 2013	Editorial corrections and updates to the following sections: I1.1, 1.2, 1.3, 2.1, 3.2 Error Handling, 3.4.2, 3.4.3, 3.4.4, 3.6, 3.7, 4.2.1. – Android, 4.2.2 - Bandwidth Awareness, 4.2.5, 4.2.8, 4.2.9, 4.2.10	PSMC & TSG	Paul Gosden, Kamran Kordi (Deutsche Telekom AG)

Other Information

Type	Description
Document Owner	Terminal Steering Group (TSG)
Editor / Company	Paul Gosden, GSMA, Kamran Kordi (Deutsche Telekom AG)

It is our intention to provide a quality product for your use. If you find any errors or omissions, please contact us with your comments. You may notify us at prd@gsma.com

Your comments or suggestions & questions are always welcome.