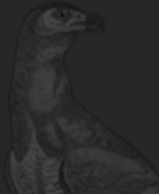




Optimizing for Mobile Networks

PERFORMANCE OF WIRELESS NETWORKS, CHAPTER 8



First off, minimizing latency through keepalive connections, geo-positioning your servers and data closer to the client, optimizing your TLS deployments, and all the other protocol optimizations we have covered are only more important on mobile applications, where both latency and throughput are always at a premium. Similarly, all the web application performance best practices are equally applicable. Feel free to flip ahead to [Primer on Web Performance](#); we'll wait.

However, mobile networks also pose some new and unique requirements for our performance strategy. Designing applications for the mobile web requires careful planning and consideration of the presentation of the content within the constraints of the form factor of the device, the unique performance properties of the radio interface, and the impact on the battery life. The three are inextricably linked.

Perhaps because it is the easiest to control, the presentation layer, with topics such as responsive design, tends to receive the most attention. However, where most applications fall short, it is often due to the incorrect design assumptions about networking performance: the application protocols are the same, but the differences in the physical delivery layers impose a number of constraints that, if unaccounted for, will lead to slow response times, high latency variability, and ultimately a compromised experience for the user. To add insult to injury, poor networking decisions will also have an outsized negative impact on the battery life of the device.

There is no universal solution for these three constraints. There are best practices for the presentation layer, the networking, and the battery life performance, but frequently they are at odds; it is up to you and your application to find the balance in your requirements. One thing is for sure: simply disregarding any one of them won't get you far.

With that in mind, we won't elaborate too much on the presentation layer, as that varies with every platform and type of application—plus, there are plenty of existing books dedicated to this subject. But, regardless of the make or the operating system, the radio and battery constraints imposed by mobile networks are universal, and that is what we will focus on in this chapter.

Note

Throughout this chapter and especially in the following pages, the term "mobile application" is used in its broadest definition: all of our discussions on the performance of mobile networks are equally applicable to native applications, regardless of the platform, and applications running in your browser, regardless of the browser vendor.



Preserve Battery Power



When it comes to mobile, conserving power is a critical concern for everyone involved: device manufacturers, carriers, application developers, and the end users of our applications. When in doubt, or wondering why or how certain mobile behaviors were put in place, ask a simple question: how does it impact or improve the battery life? In fact, this is a great question to ask for any and every feature in your application also.

Networking performance on mobile networks is inherently linked to battery performance. In fact, the physical layers of the radio interface are specifically built to optimize the battery life against the following constraints:

- Radio use at full power can drain a full battery in a matter of hours.
- Radio power requirements are going up with every wireless generation.
- Radio is often second in power consumption only to the screen.
- Radio use has a nonlinear energy profile with respect to data transferred.

With that in mind, mobile applications should aim to minimize their use of the radio interface. To be clear, that is not to say that you should avoid using the radio entirely; after all we are building connected applications that rely on access to the network! However, because keeping the radio active is so expensive in terms of battery life, our applications should maximize the amount of transferred data while the radio is on and then seek to minimize the number of additional data transfers.

Note

Even though WiFi uses a radio interface to transfer data, it is important to realize that the underlying mechanics of WiFi, and consequently the latency, throughput, and power profiles of WiFi, when compared with 2G, 3G, and 4G mobile networks are fundamentally different; see our earlier discussion on [3G, 4G, and WiFi Power Requirements](#). Consequently, the networking behavior can and often should be different when on WiFi vs. mobile networks.

Measuring Energy Use with AT&T Application Resource Optimizer



Despite the high emphasis on optimizing energy use, most platforms currently lack the necessary tools to help developers measure and optimize their applications. Thankfully, there are third-party tools that can help, such as the free Application Resource Optimizer (ARO) toolkit developed by AT&T.

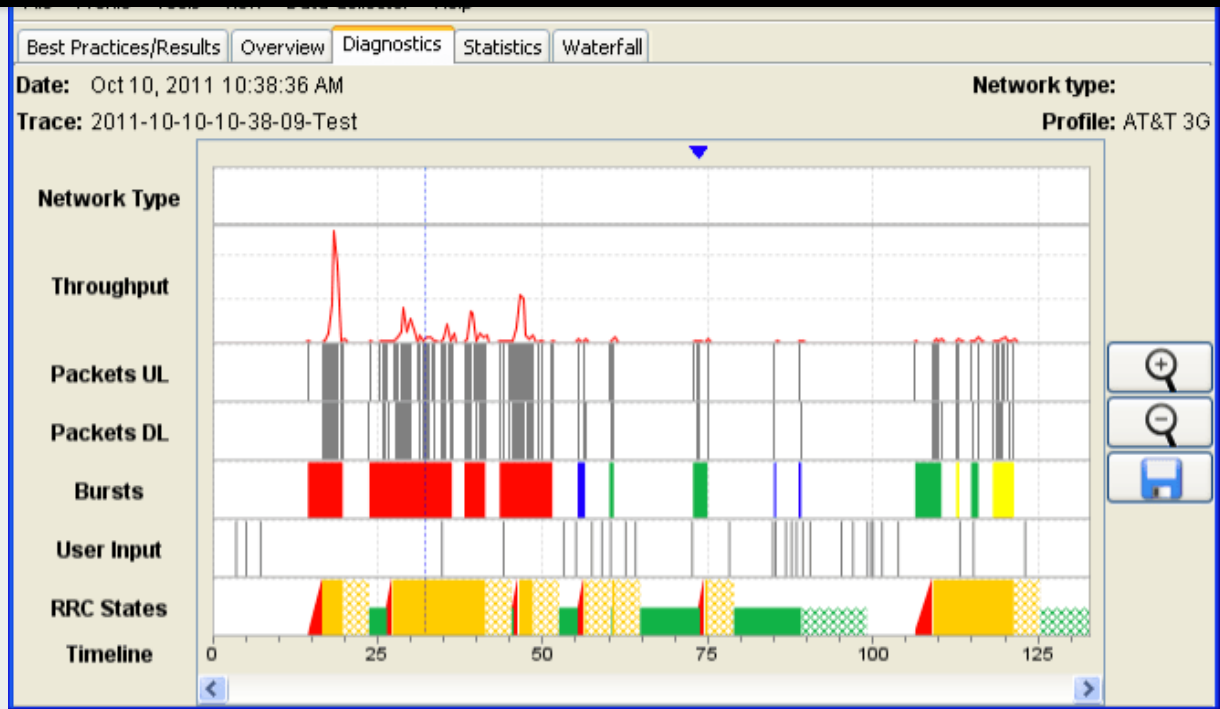


Figure 8-1. AT&T Application Resource Optimizer

ARO consists of two components: a collector and an analyzer. The collector is an Android application that runs in the background (on a real phone, or within an emulator) and captures the transferred data packets, radio activity, and other interactions with the phone. To capture a trace, load the collector, hit Record, interact with your application, and then copy the trace to your system.

Once a trace is available, you can open it with the analyzer to get insights into radio states, energy consumption, and traffic patterns of your application. One of the great features about the analyzer is that it will also provide recommendations for common performance pitfalls, such as missing compression, redundant data transfers, and more.

Two important things to note: the battery consumption and radio states are generated via a specified model of the device and the type of radio network. In other words, the generated numbers are not exact measurements from the device in use but estimates based on specified parameters in the model. On the upside, this allows you to import different device and network models and compare their energy use—e.g., 3G vs. 4G.

Finally, the collector is Android only, but the ARO analyzer can also accept any regular packet trace (pcap) file produced by tcpdump or a compatible tool; iOS users will have to use the tcpdump method.

To get started with ARO, head to <https://hpbn.co/attaro> .

Eliminate Periodic and Inefficient Data Transfers





regardless of the amount of data to be transferred, tells us that there is no such thing as a "smart request" as far as the battery is concerned. Intermittent network access is a performance anti-pattern on mobile networks; see [Inefficiency of Periodic Transfers](#). In fact, extending this same logic yields the following rules:

- Polling is exceptionally expensive on mobile networks; minimize it.
- Where possible, push delivery and notifications should be used.
- Outbound and inbound requests should be coalesced and aggregated.
- Noncritical requests should be deferred until the radio is active.

In general, push delivery is more efficient than polling. However, a high-frequency push stream can be just as, if not more, expensive. Whenever there is a need for real-time updates, you should consider the following questions:

- What is the best interval of updates and does it match user expectations?
- Instead of a fixed update interval, can an adaptive strategy be used?
- Can the inbound or outbound requests be aggregated into fewer network calls?
- Can the inbound or outbound requests be deferred until later?

Note

For push delivery, native applications have access to platform-specific push delivery services, which should be used when possible. For web applications, server-sent events (SSEs) and WebSocket delivery can be used to minimize latency and protocol overhead. Avoid polling and costly XHR techniques when possible.

A simple aggregation strategy of bundling multiple notifications into a single push event, based on an adaptive interval, user preference, or even the battery level on the device, can make a significant improvement to the power profile of any application, especially background applications, which often rely on this type of network access pattern.

Nagle and Efficient Server Push




TCP aficionados will undoubtedly recognize the request aggregation and bundling advice as Nagle's algorithm, except reimplemented at the application layer! Nagle's algorithm attempts to combine multiple small TCP messages into a single packet to reduce protocol overhead and the number of packets on the wire. Not surprisingly, our mobile applications can benefit a great deal from leveraging the same technique.

A simple implementation of such a strategy is to aggregate messages on the server by time, count, or size, instead of triggering an individual push for each one. A more involved, but significantly more efficient, strategy is to push updates only when the radio is already active on the client—e.g.,



For example, services such as Google Cloud Messaging (GCM) for Android and Chrome offer message delivery APIs which can aggregate messages and deliver updates only when the device is active: the server pushes its messages to GCM, and GCM determines the optimal delivery schedule.

Unfortunately, today there is no cross-browser API to deliver a GCM-like experience to all clients. However, the W3C Push API (see <http://www.w3.org/TR/push-api/> ) should address this use case in the future.

Intermittent beacon requests such as audience measurement pings and real-time analytics can easily negate all of your careful battery optimizations. These pings are mostly harmless on wired and even WiFi networks but carry an outsized cost on mobile networks. Do these beacons need to happen instantaneously? There is a good chance that you can easily log and defer these requests until next time the radio is active. Piggyback your background pings, and pay close attention to the network access patterns of third-party libraries and snippets in your code.

Finally, while we have so far focused on the battery, intermittent network access required for techniques such as progressive enhancement and incremental loading also carries a large latency cost due to the RRC state transitions! Recall that every state transition incurs a high control-plane latency cost in mobile networks, which may inject hundreds or thousands of extra milliseconds of latency—an especially expensive proposition for user-initiated and interactive traffic.

Calculating the Energy Cost of Background Updates



To illustrate the impact of periodic polling on battery life, let's do some simple math. The numbers are not exact but are within range of a typical 3G/4G mobile handset:

- 5 watt-hours, or 18,000 joules of battery capacity ($5 \text{ Wh} \times 3600 \text{ J/Wh}$)
- 10 joules of consumed energy to cycle radio from idle to connected and back
- 1 minute polling interval consumes 600 joules of energy per hour ($60 \times 10 \text{ J}$)
- 600 joules of energy is ~3% of total battery capacity ($600 \text{ J} / 18,000 \text{ J}$)

~3% of available battery capacity per hour for a single application! All it would take is a couple of applications with non-overlapping polling intervals to drain your battery by midday. Although, to be fair, a push application with frequent, unbuffered updates can have an even higher energy consumption profile.

Battery life optimization and frequency of updates are inherently at odds. Consider the requirements of your specific application to determine the optimal strategy: bundling of updates, adaptive update intervals, pull vs. push, and so on. Then, measure the impact with ARO or a similar tool and adjust accordingly.



The connection state and the lifecycle of any TCP or UDP connection is independent of the radio state on the device: the radio can be in a low-power state while the connections are maintained by the carrier network. Then, when a new packet arrives from the external network, the carrier radio network will notify the device, promote its radio to a connected state, and resume the data transfer.

The application does not need to keep the radio "active" to ensure that connections are not dropped. Unnecessary application keepalives can have an enormous negative impact on battery life performance and are often put in place due to simple misunderstanding of how the mobile radio works. Refer to [Physical Layer vs. Application Layer Connectivity](#) and [Packet Flow in a Mobile Network](#).

Note

Most mobile carriers set a 5–30 minute NAT connection timeout. Hence, you may need a periodic (5 minute) keepalive to keep an idle connection from being dropped. If you find yourself requiring more frequent keepalives, check your own server, proxy, and load balancer configuration first!

Anticipate Network Latency Overhead



A single HTTP request for a required resource may incur anywhere from hundreds to thousands of milliseconds of network latency overhead in a mobile network. In part, this is due to the high roundtrip latencies, but we also can't forget the overhead ([Figure 8-2](#)) of DNS, TCP, TLS, and control-plane costs!

RRC negotiation 50-2500 ms	DNS lookup 1 RTT	TCP handshake 1 RTT	TLS handshake 1-2 RTTs	HTTP request 1-n RTTs
-------------------------------	---------------------	------------------------	---------------------------	--------------------------

Figure 8-2. Components of a "simple" HTTP request

In the best case, the radio is already in a high-power state, the DNS is pre-resolved, and an existing TCP connection is available: the client may be able to reuse an existing connection and avoid the overhead of establishing a new connection. However, if the connection is busy, or nonexistent, then we must incur a number of additional roundtrips before any application data can be sent.

To illustrate the impact of these extra network roundtrips, let's assume an optimistic 100 ms roundtrip time for 4G and a 200 ms roundtrip time for 3.5G+ networks:

Control plane	200–2,500 ms	50–100 ms
DNS lookup	200 ms	100 ms
TCP handshake	200 ms	100 ms
TLS handshake	200–400 ms	100–200 ms
HTTP request	200 ms	100 ms
Total latency overhead	200–3500 ms	100–600 ms

Table 8-1. Latency overhead of a single HTTP request

The RRC control-plane latency alone can add anywhere from hundreds to thousands of milliseconds of overhead to reestablish the radio context on a 3G network! Once the radio is active, we may need to resolve the hostname to an IP address and then perform the TCP handshake—two network roundtrips. Then, if a secure tunnel is required, we may need up to two extra network roundtrips (see [TLS Session Resumption](#)). Finally, the HTTP request can be sent, which adds a minimum of another roundtrip.

We have not accounted for the server response time or the size of the response, which may require several roundtrips, and yet we have already incurred up to half a dozen roundtrips. Multiply that by the roundtrip time, and we are looking at entire seconds of latency overhead for 3G, and roughly half a second for 4G networks.

Account for RRC State Transitions

§

If the mobile device has been idle for more than a few seconds, you should assume and anticipate that the first packet will incur hundreds, or even thousands of milliseconds of extra RRC latency. As a rule of thumb, add 100 ms for 4G, 150–500 ms for 3.5G+, and 500–2,500 ms for 3G networks, as a one-time, control-plane latency cost.

The RRC is specifically designed to help mitigate some of the cost of operating the power-hungry radio. However, what we gain in battery life is offset by increases in latency and lower throughput due to the presence of the various timers, counters, and the consequent overhead of required network negotiation to transition between the different radio states. However, the RRC is also a fact of life on mobile networks—there is no way around it—and if you want to build optimized applications for the mobile web, you must design with the RRC in mind.

A quick summary of what we have learned about the RRC:

- RRC state machines are different for every wireless standard.
- RRC state machines are managed by the radio network for each device.



- RRC state demotions to lower power occur on network-configured timeouts.
- (4G) LTE state transitions can take 10 to 100 milliseconds.
- (4G) HSPA+ state transitions are competitive with LTE.
- (3G) HSPA and CDMA state transitions can take several seconds.
- Every network transfer, no matter the size, incurs an energy tail.

We have already covered why preserving battery is such an important goal for mobile applications, and we have also highlighted the inefficiency of intermittent transfers, which are a direct result of the timeout-driven RRC state transitions. However, there is one more thing you need to take away: if the device radio has been idle, then initiating a new data transfer on mobile networks will incur an additional latency delay, which may take anywhere from 100 milliseconds on latest-generation networks to up to several seconds on older 3G and 2G networks.

While the network presents the illusion of an always-on experience to our applications, the physical or the radio layer controlled by the RRC is continuously connecting and disconnecting. On the surface, this is not an issue, but the delays imposed by the RRC are, in fact, often easily noticeable by many users when unaccounted for.

Decouple User Interactions from Network Communication



A well-designed application can *feel fast* by providing instant feedback even if the underlying connection is slow or the request is taking a long time to complete. Do not couple user interactions, user feedback, and network communication. To deliver the best experience, the application should acknowledge user input within hundreds of milliseconds; see [Speed, Performance, and Human Perception](#).

If a network request is required, then initiate it in the background, and provide immediate UI feedback to acknowledge user input. The control plane latency alone will often push your application over the allotted budget for providing instant user feedback. Plan for high latencies—you cannot "fix" the latency imposed by the core network and the RRC—and work with your design team to ensure that they are aware of these limitations when designing the application.

Design for Variable Network Interface Availability



Users dislike slow applications, but broken applications, due to transient network errors, are the worst experience of all. Your mobile application must be robust in the face of common networking failures: unreachable hosts, sudden drops in throughput or increases in latency, or outright loss of connectivity. Unlike the tethered world, you simply cannot assume that once the connection is established, it will remain established. The user may be on the move and may enter an area with high amounts of interference, many active users, or plain poor coverage.



application just for the latest-generation mobile networks. As we have covered earlier ([Building for the Multigeneration Future](#)), even users with the latest handsets will continuously transition between 4G, 3G, and even 2G networks based on the continuously changing conditions of their radio environments. Your application should subscribe to these interface transitions and adjust accordingly.

Note

The application can subscribe to navigator.onLine notifications to monitor connection status. For a good introduction, also see Paul Kinlan's article on HTML5Rocks: [Working Off the Grid with HTML5 Offline](#) .

Change is the only constant in mobile networks. Radio channel quality is always changing based on distance from the tower, congestion from nearby users, ambient interference, and dozens of other factors. With that in mind, while it may be tempting to perform various forms of bandwidth and latency estimation to optimize your mobile application, the results should be treated, at best, as transient data points.

Note

The iPhone 4 "antennagate" serves as a great illustration of the unpredictable nature of radio performance: reception quality was affected by the physical location of your hand in regards to the phone's antenna, which gave birth to the infamous "You're holding it wrong."

Latency and bandwidth estimates on mobile networks are stable on the order of tens to hundreds of milliseconds, at most a second, but not more. Hence, while optimizations such as adaptive bitrate streaming are still useful for long-lived streams, such as video, which is adapted in data chunks spanning a few seconds, these bandwidth estimates should definitely not be cached or used later to make decisions about the available throughput: even on 4G, you may measure your throughput as just a few hundred Kbit/s, and then move your radio a few inches and get Mbit/s+ performance!

Streaming Applications on Mobile Networks



Streaming applications on mobile networks are a tricky problem. If you need to perform a large download and have confidence that the entire file will be used, then you should download the entire file in one shot and then leave the radio idle for as long as possible—e.g., the behavior of music file downloads in the Pandora application we covered earlier.

However, if you cannot stream the full file (e.g., an HD video) due to size or user behavior constraints, then you should leverage adaptive bitrate streaming to continuously adjust to changes



End-to-end bandwidth and latency estimation is a hard problem on any network, but doubly so on mobile networks. Avoid it, because you will get it wrong. Instead, use coarse-grained information about the generation of the network, and adjust your code accordingly. To be clear, knowing the generation or type of mobile network does not make any end-to-end performance guarantees, but it does tell you important data about the latency of the first wireless hop and the end-to-end performance of the carrier network; see [Latency and Jitter in Mobile Networks](#) and [Table 7-6](#).

Finally, throughput and latency aside, you should plan for loss of connectivity: assume this case is not an exception but the rule. Your application should remain operational, to the extent possible, when the network is unavailable or a transient failure happens and should adapt based on request type and specific error:

- Do not cache or attempt to guess the state of the network.
- Dispatch the request, listen for failures, and diagnose what happened.
- Transient errors will happen; plan for them, and use a retry strategy.
- Listen to connection state to anticipate the best request strategy.
- Use a backoff algorithm for request retries; do not spin forever.
- If offline, log and dispatch the request later if possible.
- Leverage HTML5 AppCache and localStorage for offline mode.

Note

With the growing adoption of HetNet infrastructure, the frequency of cell handoffs is set to rise dramatically, which makes monitoring your connection state and type only more important.

Burst Your Data and Return to Idle



Mobile radio interface is optimized for bursty transfers, which is a property you should leverage whenever possible: group your requests together and download as much as possible, as quickly as possible, and then let the radio return to an idle state. This strategy will deliver the best network throughput and maximize battery life of the device.

Note

The only accurate way to estimate the network's speed is, well, to use it! Latest-generation networks, such as LTE and HSPA+, perform dynamic allocation of resources in one-millisecond



much data as you can, and let the network do the rest.

An important corollary is that progressive loading of resources may do more harm than good on mobile networks. By downloading content in small chunks, we expose our applications to higher variability both in throughput and latency, not to mention the much higher energy costs to operate the radio. Instead, anticipate what your users will need next, download the content ahead of time, and let the radio idle:

- If you need to fetch a large music or a video file, consider downloading the entire file upfront, instead of streaming in chunks.
- Prefetch application content and invest in metrics and statistical models to help identify which content is appropriate to download ahead of time.
- Prefetch third-party content, such as ads, ahead of time and add application logic to show and update their state when necessary.
- Eliminate unnecessary intermittent transfers. See [46% of Battery Consumption to Transfer 0.2% of Total Bytes](#).

Building and Evaluating a Prefetch Model



Content pre-fetching will *always* create a natural tension: on one hand, you want to download as few bytes as possible, and on the other you want to minimize your latency and throughput variability and reduce your impact on the battery. Which is more important? Wrong question. The answer is always contextual to your application and the metric you choose to use to determine the effectiveness of your pre-fetch strategy.

The important takeaway is that there are, at a minimum, three variables to balance: number of transferred bytes, impact on the battery, and variability in network throughput and latency. Further, as we saw, these variables are not exclusive: transferring a larger batch of bytes in a single transfer may give you better throughput.

An application with a highly predictable usage pattern can make use of aggressive pre-fetching, minimize battery consumption, improve user experience, and simultaneously avoid incurring a large byte overhead. Conversely, a poorly implemented pre-fetch strategy can download a lot of unnecessary data and hurt the overall experience for the user.

To determine how your application should behave, first determine your primary goals and the primary usage patterns of your application. Then use that data to implement a pre-fetch strategy and gather metrics to validate the assumptions of your model. Rinse, lather, and repeat.

Offload to WiFi Networks





originate indoors, and frequently in areas with WiFi connectivity within reach. Hence, while the latest 4G networks may compete with WiFi over peak throughput and latency, very frequently they still impose a monthly data cap: mobile access is metered and often expensive to the user. Further, WiFi connections are more battery efficient (see [3G](#), [4G](#), and [WiFi Power Requirements](#)) for large transfers and do not require an RRC.

Whenever possible, and especially if you are building a data-intensive application, you should leverage WiFi connectivity when available, and if not, then consider prompting the user to enable WiFi on her device to improve experience and minimize costs.

Apply Protocol and Application Best Practices



One of the great properties of the layered architecture of our network infrastructure is that it abstracts the physical delivery from the transport layer, and the transport layer abstracts the routing and data delivery from the application protocols. This separation provides great API abstractions, but for best end-to-end performance, we still need to consider the entire stack.

Throughout this chapter, we have focused on the unique properties of the physical layer of mobile networks, such as the presence of the RRC, concerns over the battery life of the device, and incurred routing latencies in mobile networks. However, on top of this physical layer reside the transport and session protocols we have covered in earlier chapters, and all of their optimizations are just as critical, perhaps doubly so:

- [Optimizing for TCP](#)
- [Optimizing for UDP](#)
- [Optimizing for TLS](#)

Minimizing latency by reusing keepalive connections, geo-positioning servers and data closer to the client, optimizing TLS deployments, and all the other optimizations we outlined earlier are even more important on mobile networks, where roundtrip latencies are high and bandwidth is always at a premium.

Of course, our optimization strategy does not stop with transport and session protocols; they are simply the foundation. From there, we must also consider the performance implications of different application protocols (HTTP/1.0, 1.1, and 2), as well as general web application best practices—keep reading, we are not done yet!

[« Back to the Table of Contents](#)

