From Feast to Famine: Managing mobile network resources across environments and preferences

Robert Kiefer, Erik Nordström, and Michael J. Freedman
Princeton University

Abstract

Mobile devices regularly move between feast and famine—environments that differ greatly in the capacity and cost of available network resources. Managing these resources effectively is an important aspect of a user's mobile experience. However, preferences for resource management vary across users, time, and operating conditions, and user and application interests may not align. Furthermore, today's mobile OS mechanisms are typically coarse-grained, inflexible, and scattered across system and application settings. Users must adopt a "one size fits all" solution or micro-manage their devices.

This paper introduces Tango, a platform for managing network resource usage through a programmatic model that expresses user and app interests ("policies"). Tango centralizes policy expression and enforcement in a controller process that monitors device state and adjusts network usage according to a user's (potentially dynamic) interests. To align interests and leverage app-specific knowledge, Tango uses a constraint model that informs apps of network limitations so they can optimize their usage. We evaluate how to design policies that account for data limits, user experience, and battery life. We demonstrate how Tango improves individual network-intensive apps like music streaming, as well as conditions when multiple apps compete for limited resources.

1 Introduction

Even though resource management is a principle task of operating systems, mobile OSes have been slow to tackle the unique resource challenges facing mobile devices. Devices are restricted by myriad factors, including energy, computing, and network data limits. Utilizing the network tends to trade off these (usually) limited resources. Some of these—such as battery life or monthly network data limits—must be managed over epochs many orders of magnitude greater than those relevant to traditional packet scheduling. The mismanagement of the network can be costly and undesirable.

Unfortunately, today's mobile OSes offer only a limited set of mechanisms to manage resources across users, operating conditions and longer time horizons, typically in the form of hard limits. For example, to improve battery life, Apple iOS and Windows Phone limit which apps can run in the background and their network usage. Default Android supports setting simple hard data limits on the device, and some manufacturers introduce power-saving modes [1] that aggressively shut off background tasks and network usage, even if it hurts app functionality. Finally, some apps include additional settings to manage resource usage. These approaches lack sufficient flexibility and ease-of-use, as they are either too restrictive or require users to constantly micro-manage their device. Indeed, Table 1 shows the myriad of app-specific options users face to manage their network usage.

Even if third-party applications are well behaved and expose options to manage network usage, an application's behavior may not always align with its user's interests. Apps commonly focus on high performance to ensure a good user experience, rather than minimizing network usage and without concern of their impact on concurrently running apps. On the other hand, while some apps allow users to disable cellular usage or downgrade to lower bitrates, these degraded user experiences may be unnecessary given a device's current data usage.

To simplify and better align a user's interests with their device's network management, this paper introduces Tango, a system for managing network usage via a flexible, programmatic policy model. Tango centralizes network management in a controller process that monitors device state and applies dynamically-generated rules that incorporate both user and app-specific needs. This controller both distills ad-hoc user configuration into a single user policy and provides a means of resolving conflicts between user and app policies. Towards this end, user interests trump app interests in Tango, and a constraint mechanism is used that enforces limits (even for legacy or uncooperative apps) while *encouraging* apps to align. The constraint system *proactively* deals with policy conflicts, informing apps of their limits so they can adjust. Alternative *reactive* solutions, such as discarding or changing outputs of conflicting policies, would leave apps either uninformed if their policy was carried out or unable to know what policies are acceptable in the first place. Meanwhile, cooperative apps can leverage the constraints and local knowledge (e.g., about buffers or counters) to optimize their usage. Additionally, Tango allows apps to "hint" about their needs (e.g., priority, data rate) that become part of device state. These hints can be incorporated by the user policy, allowing apps to provide feedback while still maintaining user policy preference.

A Tango *policy* is a programmatic instantiation of a user's (or app's) interests. Given the device state and constraints, a policy outputs a list of actions to take. Because mobile devices' operating conditions are quite dynamic, Tango collects information from a variety sources—including the kernel, network stack, and battery—to be used as input to policies. For example, a user may be interested in using WiFi over HSPA, except when it provides unacceptable throughput. When considering whether to use WiFi or HSPA, a policy can consider the number of apps using the network, whether there is any foreground activity, and the current traffic demands on the network. This model is considerably richer and more flexible that the options currently available.

To demonstrate Tango's use, we apply it to the context of both single apps and the concurrent use of network-competing apps. For the former, we investigate WiFi offloading for a music streaming app in areas of spotty WiFi coverage, and find Tango can help achieve both performance (i.e., seamless playback) and data savings for the user (60-97%). For the latter, we show how policy can be used to express fairness and prioritization across apps. We demonstrate how user policy can provide per-app fairness regardless of the number of flows in each app, as well as how user policy can prioritize usage based on changing operating and app conditions.

2 Motivation and Challenges

Today, music streaming services, like Pandora, Rdio, Spotify, Google Play Music, and iTunes Radio, are popular alternatives to preloading devices with large music libraries. In fact, a recent study [22] points out that media streaming (including music) is one of the major sources of cellular data usage. Using music streaming as a motivating example, we next highlight some areas in which current network management does not afford a good user experience. We also discuss challenges in aligning the interests of users, device vendors, and app developers.

2.1 Balancing Costs, Caps, and Battery

With unlimited data plans a thing of the past, the increased flexibility of streaming comes with the risk of inflating cellular data usage and power consumption. Prefetching upcoming songs to provide a seamless playback experience users expect inevitably leads to trade-offs, where the data limited and power hungry cellular link must be used.

App class	App name	Settings
Social	Facebook	Refresh rate: None / 30m / 1h / 2h / 4h Sync photos: Any network / WiFi only
	Google+	Sync photos: Any network / WiFi only Sync videos: Any network / WiFi only Sync while roaming: On / Off Sync only while charging: On / Off
Audio Streaming	Pandora	High-quality on cell: On / Off Conserve battery: On / Off
	Google Play Music	Cache during playback: Yes / No Auto cache while charging+Wifi: Yes / No Pin songs on WiFi only: Yes / No Stream on WiFi only: Yes / No Cell stream quality: Low / Normal / High
Video Streaming	Youtube	HD on cell: Yes / No Uploads: Any network / WiFi only Preload WiFi+Charging: None / Subscriptions / Watch Later / Both
	Netflix	Playback on WiFi only: Yes / No

Table 1: Some of the myriad application settings available for managing resource usage.

Table 1 shows that apps try to address managing these trade-offs by providing many knobs for the user to tune. However, this forces the user to choose reduced functionality or constant micro-management. The level of control exposed is left up to app developers, making it inconsistent even across apps of the same class (e.g., Pandora vs. Google Play Music). Further, some app settings assume a certain level of understanding of the inner workings of the app, and settings may come and go as the app is updated. Finally, a user's interests can change over the course of a billing cycle depending on their data use, or over the course of the day as their battery drains. Trying to balance all these concerns, via numerous, redundant, inconsistent settings and over an entire billing cycle, becomes an insurmountable problem for many users. With Tango, many of these settings are distilled into policies that are configurable from a single location via the user policy. Still, Tango supports app-specific settings, but may override them if they conflict with the user policy.

2.2 Ensuring Good User Experiences

Smartphones automatically switch between networks (e.g., 4G and WiFi), but this can be disruptive to the user experience. Network switching is done with little regard to its effect on apps, which typically experience failed TCP connections when IP addresses change. For streaming music, this manifests as pauses in playback; other apps can see broken webpages, disconnects from a game, or other failures. While every app could deploy its own failure handling, this leads to little economy of mechanism and places an undue burden on developers. Further, efficient recovery is not always supported by remote servers (e.g., only about 39% of the top ten thousand Alexa websites supports HTTP range requests [14]). In the worst case, we found some popular streaming apps that simply fail and tell the user to try again.

This problem is exacerbated by the aggressive WiFi offloading performed by today's smartphones, even when performance on WiFi is worse than cellular. Examples of this include public hotspots that are overloaded with too many users, networks with weak signal, or networks with a low-bandwidth backhaul link. Without a way to seamlessly switch between networks and an effective policy for when to switch, the user is left to either manually manage their connectivity, live with the poor performance, or inflate cellular usage by never using WiFi. Tango improves the user experience by allowing data streams (e.g., music playback) to continue seamlessly across network changes (by adopting MPTCP [21] or Serval [12]) and having policies dynamically pick the best network according to user interests.

2.3 Managing Conflicting Interests

Conflicting user and app interests are a further source of tension on mobile devices. Apps typically focus on ensuring they perform well, even at the expense of wasted data usage. This is particularly a problem with streaming apps, e.g., where prefetched content may be discarded when the user skips songs, stops playing, or reaches WiFi before it is needed.

Interest conflicts occur among concurrently running apps. Some mobile OSes, like iOS or Windows Phone, attempt to minimize these conflicts by limiting the types of background apps (e.g., media streaming, location tracking). Yet such coarse-grain policy inhibits non-whitelisted apps, while still failing to provide sufficient resource isolation. For example, streaming apps will prefetch songs even if their need is not imminent, slowing down interactive foreground tasks such as web browsing. Also, because an app's usage may be spread over many flows—e.g., one for streaming media and another for prefetching—traditional resource-management policies like per-flow fairness may be insufficient. Further, resource prioritization can be more nuanced than a mere differentiation between foreground and background apps. For instance, background music streaming's data usage is as important as the foreground task when its buffer is low, but less important as the buffer fills up.

Tango supports the restriction and prioritization of resources on a per-app basis. By considering apps rather than flows, it prevents any "strength-in-numbers" attempts by apps to gain an unwarranted portion of network resources. User policy can dynamically prioritize apps on their execution status, as well as consider hints from apps about when to re-prioritize their usage (e.g., when a background music app signals the need for higher priority when its buffer fell below a low watermark).

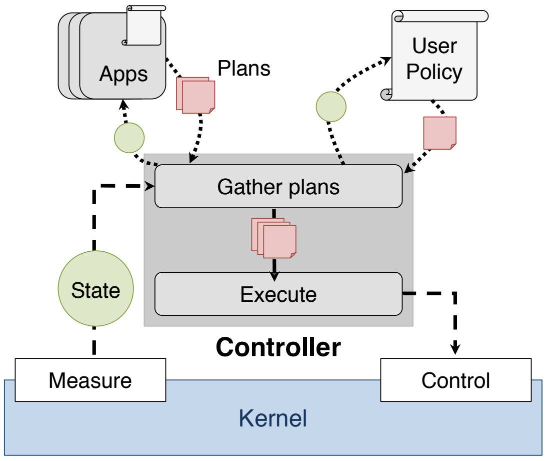


Figure 1: The Tango architecture

3 Tango Design

This section details Tango's design and how it manages network usage. Figure 1 shows a high-level overview of Tango's architecture. At the heart of Tango is a controller that runs as a privileged process, used to centralize network control and to handle the dynamic nature of mobile devices. The controller is responsible for monitoring and packaging device state into a common API to be used by policies. A *policy* in Tango refers to a programmatic instantiation of a party's interests. Policies use device state to derive a *plan*, a list of actions it would like taken on its behalf. These plans are vetted by the controller to make sure they are valid, including that they obey *constraints* set by a higher-priority policy (i.e., from the user). Constraints serve as a mechanism to proactively resolve interest conflicts between a user and an app (or between apps), providing limitations on usage by an app (e.g., "rate limit of 200 kbps on HSPA"). In the rest of this section, we detail the responsibilities and benefits of the controller, the role of user and app policies, and look at policy in practice. Throughout this section we include pseudocode examples to give a sense of what policies look like and how simply many of these concepts can be expressed in our framework. While we use pseudocode here, the actual policies are very similar in terms of complexity and length; the pseudocode is just more terse than the language in which are prototype policies are written (Java, see §4).

3.1 The Controller and Policy Execution

The controller process is at the heart of Tango, designed to be a central point for network management. The controller's control loop is presented in Pseudocode 1. This loop is run once per control epoch, which is configurable and typically on the order of seconds. Alternatively the loop could run in an event-based manner,

Pseudocode 1 Tango Control Loop

```
1: A: applications
2: for every epoch do
3:     S ← MeasureDeviceState
4:     C ← DetermineConstraints(S)
5:     Enforce(C)
6:     for a ∈ A do
7:         SolicitPlan(a, C)
8:     P ← GatherPlans(A)
9:     for p ∈ P do
10:        if not Valid(p, C) then
11:            p ← GetDefaultPlan
12:        Execute(p)
```

Network stack	
Transport	# retransmissions, RTTs, congestion window, etc.
Network (IP)	addresses, routing rules, etc.
Physical	link type, signal quality, bit errors, etc.
Other sensors	
Battery	plugged in, charge percent, current draw, etc.
GPS	location, speed

Table 2: Device state sources and metrics.

responding to device state changes like new interfaces becoming available, particular apps opening, etc.

Monitoring device state. The control loop starts with the controller compiling a current view of the device state, which it exposes as a high-level API to policies. This provides apps with a common way to access device state, rather than leaving the implementation up to each app. The device state is composed of metrics from numerous sources including the OS, the network stack, battery, apps, and optionally other available sensors, as shown in Table 2. Reading some of these sources may require elevated privileges, so by centralizing this process we also remove the need for apps to request additional permissions. Further, Tango enforces information protection by only sharing relevant state with an app, e.g., the transport-layer statistics for its flows.

This device monitoring stage is important for handling the dynamic environments in which mobile devices are used, as it allows the controller to stay up-to-date with operating conditions. While current network management mechanisms recognize the importance of certain state (e.g., WiFi signal strength), Tango increases the scope of monitored state; bandwidth, latency, app foreground status, etc., provide a more complete picture of the current environment. For example, in the case of overcrowded public hotspots, available bandwidth is a much more important metric for quality than signal strength. The information supplied by Tango allows for more versatile and useful policies.

Enforcing constraints. One task of the user policy is to specify constraints to the system. Constraints are limits on system resources that are enforced on entities such as interfaces and apps. For example, to reduce data usage, a user policy could constrain the HSPA interface to 500 kbps, which the controller applies using system tools and APIs (§4). The controller rejects actions generated by policies attempting to violate this, e.g., an app trying to set its limit to 1Mbps. In this manner, a user's interests are enforced even in the presence of misbehaving apps.

Tango uses this constraint mechanism to proactively resolve conflicts between competing interests. In Tango, user interests trump those of apps, which is why the user policy determines these constraints. Constraints are intended to allow app policies to cooperate with user policy, but are also always enforced by the controller, even if a malicious app policy tries to skirt them. By telling app policies of constraints upfront, apps can optimize their usage within those limits. For example, a user may rate limit a streaming app to marginally above its playback rate; the app adjusts by de-prioritizing non-critical flows (i.e., prefetching) in favor of critical ones (i.e., streaming). If a user's constraints cause the app to perform poorly, either the user policy needs to be refined to be less restrictive, or the app's interests are too divergent. We imagine that frequently conflicting policies can hurt an app's online feedback, incentivizing apps to include policies that work well across operating conditions.

We preferred this proactive approach, with its straightforward controller task of approving app plans, over attempting to resolve conflicts after app policies had been evaluated. To do that, conflicting plans would need to be modified by the controller or use a feedback loop that involves reevaluating app policies. The former leaves apps optimizing for conditions that may not happen, while the latter also need a constraint mechanism to avoid the repeating conflicts in subsequent rounds. Additionally, we chose to make user policy trump app policy because the device belongs to the user and *they* will ultimately "pay" the consequences of resource mismanagment (e.g., data overage costs, the battery dying before the end of the day). From this standpoint, apps that fail to comply with user policy can be thought of as *violating correctness*. App hints and policies allow some room for compromise, however, within the confines specified by the user policy's constraints.

Policy conflicts also occur across different apps. In such situations, constraints are useful for expressing prioritization and how usage should be shared amongst the conflicting apps. Since a constraint applies on a per-app basis, rather than per-flow, it is not possible for apps to game the system by a "strength in numbers" approach. That is, creating multiple flows to gain more bandwidth is futile. App priority can be specified by allocating larger resource shares to high-priority apps, or by limiting lower-priority apps while leaving others unrestricted.

Pseudocode 2 App policy with hints

```
PS: music player state
urgent: urgent need for data
function EVALUATE(S, C)
    P: plan
    // Rest of policy elided for space.
    P.hintPriority ← NORMAL
    if PS.getBufferTime() >30 then
        urgent ← false
    else if PS.getBufferTime() <20 || urgent then
        urgent ← true
        P.hintPriority ← HIGH
    return P
```

Pseudocode 3 User policy with app hints

```
function DETERMINECONSTRAINTS(S)
    C: constraints
    for A in S.apps() do
        if ALLOWPRIORITY(A, A.hintPriority) then
            C ← NEWAPPCONSTRAINT(A, A.hintPriority)
        else
            C ← NEWAPPCONSTRAINT(A, NORMAL)
    return C
```

Supporting app policies. Tango allows apps to specify their own policies. During each epoch (lines 6-8 in Pseudocode 1), the controller disperses appropriate device state and constraints to apps registered with the controller. The app policy responds with a plan, which the controller validates (lines 9-12), ensuring the app only manages its own flows or other approved resources and obeys its given constraints. The user policy provides a default plan for apps without one or an invalid one.

App policies allows apps to refine their network usage based on local information that a user policy would not know. For example, if a user policy restricts an app's data limit, the app policy can respond by asking the controller to re-prioritize certain flows higher (e.g., those downloading a social network feed) than others (e.g., background syncs). In this way, apps are given more insight into what is happening to their network usage and given a way to respond in useful ways. Should an app not provide a policy, the typical default plan by user policies would be a bare minimum approach. That is, enforce the constraints on that app, but little else.

In addition to actions to take on its behalf, apps can provide hints to the controller about its needs. For example, when a music app's playback buffer is low, it can send a hint that it wants higher priority for its traffic. This information is stored as part of the device state for the next control epoch, which can be used by the user policy as part of its constraint generation process. Pseudocode 3 and 2 are examples of user and app policies using app hints. The app policy sets `hintPriority` in its plan. In the next epoch, the user policy uses `AllowPriority()` to decide if the app's request is allowable, e.g., by matching against a list of apps and their acceptable priority levels. Apps therefore can provide feedback to the user policy, which still has the final approval of what hints to use. We look more at the usefulness of app hints when it comes to app conflicts in §5.3.3.

3.2 A Programmatic Approach to Policy

Tango employs policy at two levels: the user (device) and the application. User policies reflect the overall desires of the device owner, which may reflect high-level interests such as "preserve battery," "minimize cellular usage," or "ensure high throughput for video." Management of a device's interfaces, and how usage is shared or prioritized across different (classes of) apps, are expressed by the user policy. User policies can naturally be written with classes of apps (e.g., music streaming apps) in mind; the class of each app can be prepopulated by the policy writer, configured by the user in settings, or suggested by the app developer. This allows common constraints and goals (e.g., reduce data usage) to be set once for several apps, and reduces the complexity of policies by not focusing on individual apps. There is only one user policy running at a time, and it cannot be changed by third-party apps.

We imagine there are a few ways for users to select a policy. Device manufacturers, or even tech-savvy users themselves, could write user policies, which can then be configured and activated in the device settings. Policies could be shared by uploading them to a "policy store," where other users could download and review them. When a user loads a policy, it can expose configuration options that users can tune to fit their needs (e.g., monthly data budget, desired music quality, etc.).

App policies, on the other hand, allow the system to reflect the needs to currently executing applications. They may use information only visible or semantically meaningful to the app, e.g., a streaming app's policy examines the playback buffer when expressing "do not buffer when above X seconds." By soliciting plans from apps, the controller can account for such app-specific information in its control loop, provided it does not violate the constraints set by the user policy. Apps can only specify policy which affects their usage; they can not manipulate usage of other apps or set the user policy. In our experience (§5), adding policy to apps was straightforward and a matter of exposing the pertinent information (e.g., buffer levels) to the app policy via shared state. Also, since the actions are handled by the controller, there were less permissions and code needed for the app itself.

Implementing and expressing policies. Policies are programs that implement a simple interface. This interface consists of an `evaluate()` function that constructs a plan (a list of actions) given (1) the current device state, e.g., network metrics, available interfaces, battery life, (2) constraints such as bandwidth limits, and

Action	Iface	Flow	Description
ENABLE	✓	✓	Enable iface/subflow
RATELIMIT	✓	✓	Limit bandwidth
LOG	✓	✓	Write information to file
MANAGE	✓		Change AP, queue size, etc
MIGRATE		✓	Move flow to new interface

Table 3: Actions on interfaces and flows. App policy can only perform flow actions and only on their flows.

Pseudocode 4 Avoid poor WiFi

```
sigs: list of WiFi signals
slowNets: map networks to time added
function EVALUATE(S, C)
    P: plan
    wifi ← GETWIFIINTERFACE(S)
    cell ← GETCELLINTERFACE(S)
    if wifi.isAssociated() then
        sigs.push(wifi.signal())
        if BADSIGNAL( )then
            P.add(MANAGE, DISCONNECT, wifi)
            P.add(MANAGE, CONNECT, cell)
        else if wifi.speed() <100000 then
            slowNets.put(wifi.network(), S.now())
            P.add(MANAGE, DISCONNECT, wifi)
            P.add(MANAGE, CONNECT, cell)
    // Other cases elided for space.
    return P
```

(3) a list of controllables. Additionally, the user policy's `determineConstraints()` function returns interface and app constraints based on the current device state.

A plan is a list of actions that a policy would like executed. Tango currently exposes two types of controllable entities: *interfaces* and *flows*. Actions on interfaces include turning interfaces on and off, selecting access points (APs), and setting queue and rate limits. Interface actions are only available to the user policy, as they affect all apps on the device. Actions on (sub)flows include adding/removing flows and migrating them across interfaces. These actions are available to both user and app policies, though app policies are restricted to acting only on their own flows. Table 3 summarizes these actions.

Tango's programmatic approach provides sufficient flexibility. It allows for simple, rule-based approaches (like current techniques), as well as more complicated approaches that use past behavior to predict future usage. An example policy snippet for WiFi-connected devices is given by Pseudocode 4. Two conditions cause the policy to return a plan that fails over to HSPA: (1) if `BadSignal()` return true or (2) if the measured speed of WiFi is below 100 kbps. There are many possible implementations for `BadSignal()`, including tracking a list of past signal readings to monitor trends instead of instantaneous readings (see §5.2). Condition (2) helps address several scenarios mentioned earlier, such as overcrowded hotspots. This is just one sample implementation; others could integrate information about the mix of apps using the network, location, battery life, and more.

3.3 Discussion: Policy in Practice

We now revisit the discussion of use cases from §2, and expand on how policy improves usage in practice.

From incidental to intentional device behavior. Today's mobile device network management, with settings spread across apps, has at best an *incidental* effect on resource usage. Users are not certain whether their combination of settings will translate into what they want. In contrast, Tango allows users to load and run policies that have *intentional* effects on the way a device behaves. Unlike today's myriad settings, policies express what the user wants, not how it is achieved. Tango allows control of device behavior from a single location, and settings are structured into global and class-specific ones. Global settings apply to all classes, while class settings could generate constraints for all apps of that class, such as a slider for streaming quality on cellular for all media apps. Apps may still have settings to further tune their usage, but they are subject to global constraints ensuring they are aligned with the user policy.

Improved user experience. Mobile OSes have many sensors and control surfaces that govern device behavior, but no effective way to translate that flexibility into an improved user experience. For instance, OSes that support seamless flow migration (e.g., iOS7 with MPTCP [18]) have the ability to switch networks without interrupting individual flows. However, as we have pointed out, many available networks are overloaded or experience weak signals. Unless flow migration is governed by an effective policy, this feature may have limited practical effect on user experience. Programmatic policy allows for solutions like building profiles of networks and usage over extended periods, to later inform a decision on whether to use a particular network.

Living within one's means. Tango's constraint mechanism is useful for reigning in cellular usage. For data-heavy apps such as media streaming, the user policy could restrict those (classes of) apps, either with a static rate limit or by allotting them an amount of usage over a time interval. A static rate limit is useful for curtailing usage in the case of apps that do not have an app policy, but can cause poor performance if cellular service disappears (depleting the buffer) or if the rate is set too low. Assigning an allotment is similar to a rate limit, but provides apps with greater flexibility to optimize their usage. For example, an app with a sufficient media buffer can save its allotment until hitting a low watermark or for unexpected future events that necessitate a fast response, such as the user skipping the current song.

Prioritization and fair network usage. Tango makes supporting priority and fairness in network usage straightforward with constraints and the controller's state

Pseudocode 5 Prioritize foreground app

```
1: function DetermineConstraints(S)
2:     C: constraints
3:     for A in S.apps() do
4:         if A.isForeground() then
5:             C ← NewAppConstraint(A, HIGH)
6:     return C
```

monitoring. A common example of network usage to prioritize would be that of the foreground app (changes of which are learned quickly by the controller). Achieving this prioritization in Tango can done by implementing the user policy's `DetermineConstraints()` per Pseudocode 5. It also takes care of managing the kernel traffic queues for the user and ensures it does not cause unintended side effects with other settings. Since constraints apply on a per-app basis, Tango can effectively reign in greedy or buggy apps that would otherwise drain resources, e.g., by having many open TCP connections.

4 Implementation

We have implemented a prototype of Tango that runs on Android devices. The prototype is in Java and consists of three parts: (i) the main library with code for generating device state and APIs for policies, constraints, and actions; (ii) a client library for app policy communication to the controller; and (iii) a controller with most of the previously described functionality.

Our prototype includes support for flow migration on Linux via ECCP [2], as implemented in a loadable kernel module through the Serval network stack [12]. Serval has support for getting flow-level metrics from the transport layer, such as RTT and congestion window. While we chose this particular implementation of flow migration to use with our prototype, Tango is not dependent on it. Other methods for gathering flow-level metrics [19] and flow migration [17, 21, 23] would work as well.

Our prototype leverages many resource management mechanisms already available in Linux and Android, and thus did not require any OS modifications. The `tc` tool provides rate limit actions and constraints; we use the hierarchical token bucket (HTB) queue extensively for enforcing constraints. Each interface starts with an overall bucket which can be rate limited via constraints or actions. Buckets are attached to interfaces when per-app rate limiting or prioritization is needed. We use filtering rules to put all of an app's flows on an interface into the appropriate bucket. Additional buckets can be attached to these queues to do finer-grain QoS at the flow level. Other functionality for managing networks are done with standard Linux tools (e.g., `ip`, `iptables`, etc), or via APIs in the Android SDK (e.g., WifiManager).

Along with Tango, we have implemented several user policies and a few test apps that use our client library. Our policies are written as Java classes that easily interface with our prototype and, in the case of app policies, the apps they are for. In particular, we have implemented a music streaming app that we use in our evaluation. The app downloads MP3 files over HTTP using a native Serval socket,[1] supporting near instantaneous playback of partially downloaded files. Playback starts with 30 seconds of content buffered and pauses if the buffer runs out; play resumes when the buffer again reaches 30 seconds. This functionality matches the functionality of many popular music streaming apps, such as Pandora and Google Play Music.

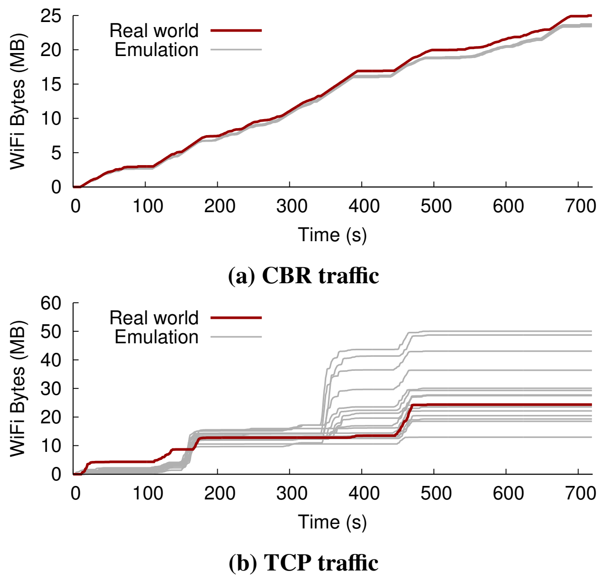


Figure 2: Emulations of a campus walk. Both CBR traffic (a) and TCP traffic (b) show high fidelity.

5 Case Study Evaluation

In this section, we perform two case studies that aim to address the following questions: First, how can Tango help improve the experience of using a single phone app? With music streaming as our example, we perform comparative studies of both user- and app-level policies in the face of changing environment conditions. Second, how can Tango help provide a good user experience across concurrently-running apps? In particular, we explore how policies enable us to provide dynamic fairness and/or QoS based on changing device conditions.

5.1 Experimental Setup

We use a Galaxy Nexus (GSM) phone running Android 4.3 for our case studies, with T-Mobile as our HSPA data provider. We installed Tango, Serval [12], and our music streaming app on the phone.

To evaluate music streaming, we wanted to use the scenario of a student walking across campus, attempting to use WiFi to save data. However, as is typical

[1] Serval has a proxying solution that allow for unmodified apps to be included in Tango's planning.

with a wireless environment, our field measurements observed highly variable coverage and quality, even on back-to-back walks. Thus, to make meaningful comparisons across policies, we set up an emulation environment created from traces of our walks and replayed on our phone. This allowed for repeatable experiments of different policies for the same walk. The traces we use are from walks where Android chose WiFi over cellular between 75-95% of the time; however, our results show only about 20% of that time is the link able to carry data. The trace covers roughly a mile across campus that took 12 minutes to complete. We connected to the campus WiFi, which uses the same SSID across many access points. Overall, WiFi coverage was mostly continuous in the middle of the walk and more spotty towards the ends. The traces were collected in the afternoon during the school year, so congestion levels were typical.

Emulation environment. We implemented the emulation using the standard Linux `tc` tool to recreate WiFi network conditions vis-a-vis packet loss and delay. The emulation leaves the portions of cellular connectivity in our traces unregulated, as we generally had no problems with cellular coverage.

To capture the variable WiFi network conditions during real walks, we used a `ping`-like application sending packets at a constant bit rate (CBR) to a server (one packet every 20 ms) with packet sizes to emulate TCP (1472-byte echo packets with 64-byte replies). We measured the received signal strength indicator (RSSI), upstream and downstream loss rates, and delay for every second of the walk. During emulation, the loss rates and delays were parameters for `tc`, while the RSSI readings replaced the readings from the actual WiFi driver.

We validated the accuracy of our emulation with two types of traffic: CBR and TCP. We used the CBR traffic to assess the connectivity and drop rates and the results are shown in Figure 2a. Since `tc` drops packets probabilistically, we ran five emulations. There was low variance in the cumulative bytes downloaded on WiFi during these emulations (one gray line per trial). The 6% difference between the real-world and the emulations are from delays going from cellular to WiFi, i.e., DHCP and cellular teardown.[2] Most notably, the "shape" of the bandwidth usage is consistent across trials.

To access the emulation's accuracy involving TCP's congestion and flow control, we used our music streaming app. The results are shown in Figure 2b, encompassing 15 emulated trials. Due to TCP's congestion control and reliable transfer mechanism, there is much greater variance in the results. Yet, the emulated trials still capture the overall "shape" of the connectivity well, i.e., they mostly share the same increases (good WiFi) and flat areas (bad WiFi). One notable difference between trials is around 350s where roughly half the trials download a significant amount of data, while the others (and real world) do not. The real world trace shows good signal quality with low drop rates and delays in that period, but long TCP timeouts started just before good connectivity cause it to be missed during some emulations (which we manually verified in our traces). Apart from some trials having "luck" in how their TCP timers fire, all trials otherwise experience similar behavior.

RSSI	Number Intervals	Time%	Up Drop%	Down Drop%	Good%
(-90, -85]	54	3.88	63.56	56.97	11.11
(-85, -80]	283	20.36	55.85	55.57	9.19
(-80, -75]	367	26.40	45.00	45.67	15.80
(-75, -70]	267	19.21	33.05	31.90	34.08
(-70, -65]	180	12.95	21.06	22.68	47.22
(-65, -60]	155	11.15	10.91	10.50	75.48
(-60, -55]	60	4.32	2.54	1.12	95.00
>-55	24	1.73	2.29	0.83	100.00

Table 4: WiFi connectivity quality statistics across many traces of the same path. "Good" signifies both upstream and downstream had drop rates $\leq$10%.

Network switching. The periods of poor WiFi connectivity that cause long TCP timeouts highlight a problem with Android's default network switching: it prioritizes WiFi too much. To this end, we developed a network switching scheme to reduce the duration on unusable WiFi. Android uses WiFi whenever the RSSI is -100 or better. Rather than using an instantaneous measure, our Tango user policy tracks the last 10 seconds worth of RSSI values (one per second) and uses two heuristics to determine if the signal is degrading sufficiently to switch: (i) all 10 RSSI values have been below -75, and (ii) whether the last five were all below -80. We chose these heuristics after analyzing multiple traces at different signal levels (see Table 4) and testing them around campus. While heuristics help determine when to move off WiFi, the move from cellular to WiFi relies on instantaneous readings based on regular WiFi scans. For this, we chose an instantaneous RSSI of -70 or greater as our threshold; -70 nearly doubles the amount of usable WiFi (30.15% vs 17.20%) and has an acceptable drop rate.

Figure 3 shows a CDF of the duration of "poor connectivity" zones on WiFi using the Android connectivity manager, our Tango switching based on the above heuristics, and the CBR baseline. "Poor connectivity" is any WiFi interval where the download rate was less than the playback rate. The CBR traffic should represent the ideal distribution since it is not subject to TCP's timeout effects. Android's switching has a very long tail, showing the extremely long periods of no data transfer caused by TCP timeouts. The bumps in the distribution appear to roughly correspond with when TCP timeout re-

[2]The emulator reacts to logged network switching events, which in the real world are initiated 1-2 seconds prior to being logged.

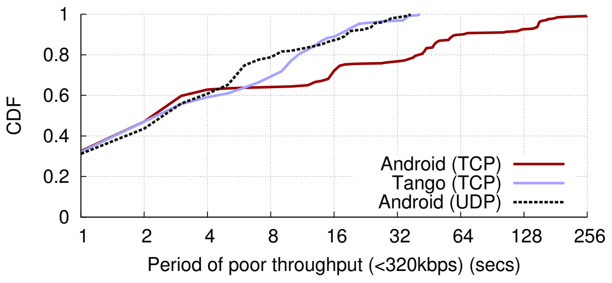


Figure 3: Distribution of durations of low TCP throughput, Android versus Tango switching. CBR UDP traffic serves as a baseline showing connectivity.

	Tango switching	Constraints	App policy
`Unl`	✓		
`Rate`	✓	✓	
`App`	✓	✓	✓

Table 5: Evaluated policies.

tries would re-establish transfer. Conversely, Tango only briefly diverges from CBR between 5 and 10s. This is most likely due to its heuristic to switch off WiFi after 10 seconds of poor RSSI. Tango's pre-emptive switching prevents long TCP timeouts by moving the flow to cellular where it can continue transferring or at least respond to probes. Aside for §5.2.1, our subsequent evaluation will use Tango's switching scheme.

Configurations and policies. In the case studies, we use our emulator with the music streaming app playing at 320 kbps. The trace we use typifies a walk through our campus in terms of its connectivity and WiFi coverage.

To evaluate Tango's use of multi-level policy, we compare both user and app policies against a baseline that allows unlimited data usage by apps, highlighting the effect of different levels of constraints and policies. The policy configurations evaluated are summarized in Table 5, where `Unl` allows unlimited rates (no constraints), `Rate` applies a rate-limiting constraint of 640 kbps (double playback rate), and `App` includes both user- and app-level policy. With `App`, the user policy allows the application to use some allotment of data over a given time frame. This constraint allows for bursts of traffic (for application buffering), rather than a constant limit. The app performs flow control using high and low watermarks; when the buffer goes below the low watermark, the app downloads until above the high watermark.

5.2 Case Study 1: Music Streaming

As discussed in §2, music streaming on mobile devices needs to balance data caps, costs, and battery life against the ability to provide a seamless and high-quality listening experience. WiFi offloading is a natural way to reduce cellular usage and avoid hitting data caps. Yet streaming has time requirements that do not always allow network usage to be deferred (i.e., the user wants to listen now, not wait until a WiFi hotspot). Thus, users are often confronted with an unfortunate trade-off between user experience and economics.

In contrast, with Tango's seamless migration of flows, phones can switch networks when appropriate, as well as defer downloading some content until when the conditions are right on WiFi. This also works with apps that do not have their own failure and recovery mechanisms. Even with failure-handling apps, the ability to put constraints on cellular usage helps reduce costs for users, and network load for wireless providers.

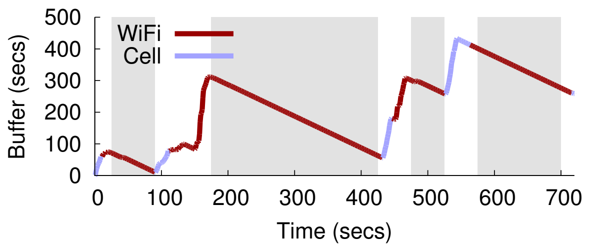
5.2.1 Effect of Aggressive WiFi Offloading

Although WiFi offloading is desirable, it does little good unless governed by a policy that determines a good time to switch interfaces. To illustrate the (negative) effect of aggressive WiFi offloading, we first show results with Android's default network switching, but with added flow-migration functionality. With this setup, flows are seamlessly migrated to the active interface.

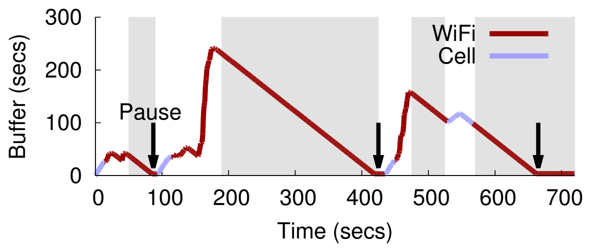
Figure 4a shows the app's buffer size (in seconds) during the emulated walk. While the app is able to play music without pauses, it does so with excessive cell usage, e.g., at time 527s. The buffers are well filled at this point, but because there are no limits in place the app downloads indiscriminately. On this particular walk, we calculated 12MB of excess data, which can add up quickly (e.g., up to 500MB monthly if part of a twice daily walk to the office). Moreover, there are long periods of poor WiFi with no TCP progress (shaded in the graph). This is worrisome for two reasons. First, apps that cannot buffer as aggressively, such as live streaming, would likely fail many times on this walk. Second, when there is clean signal in the emulation, the client and server are on the same network, which allows more buffering than in real life over more congested wide-area links.[3] Thus, with Android's default behavior, network usage is disproportional to need on cellular, and WiFi is used inefficiently.

Unfortunately, clinging to WiFi leaves little room for reducing the cellular excess as the margin of error is small. To illustrate this, we applied a rate limit to cellular that should allow continuous playback of 640 kbps (2X playback rate). Cellular usage is reduced by almost 15 MB—a median of 19.8 MB down to 5.0 MB—but introduces playback pauses, as seen in Figure 4b. Because of these problems, for the rest of our experiments, we use our heuristics-based switching scheme to significantly reduce the times spent on poor-quality WiFi.

[3]We had the server and client on the same network for a more controlled experience across emulations.

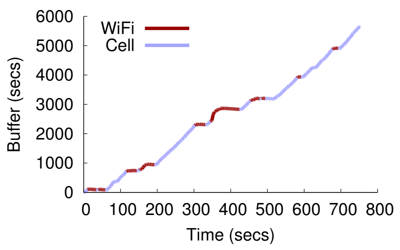


(a) Android sticks to poor quality WiFi (shaded) for extended periods, yet also needlessly downloads on cellular at times.

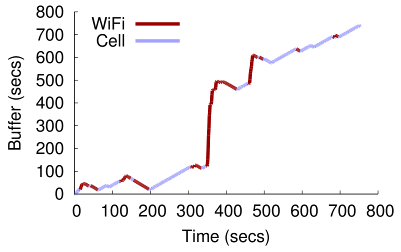


(b) With lower cellular usage, even large buffers cannot mitigate long periods of no connectivity, leading to multiple playback pauses.

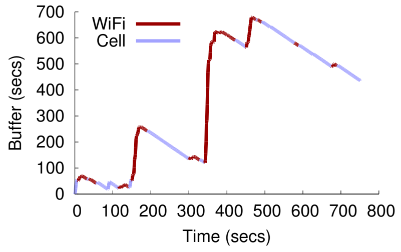
Figure 4: Effect of aggressive WiFi offloading on playback buffers. Android's tendency to persist on WiFi, despite no TCP progress, leaves little room for policy to play a role in improving the application experience.



(a) `Unl`: Buffer increases at full rate, irrespective of connectivity.



(b) `Rate`: Buffer increases at full rate on WiFi, slower rate on cellular.



(c) `App`: Buffer increases only when necessary according to app policy.

Figure 5: Buffer usage of different Tango policies. All avoid any pauses during playback.

5.2.2 Finding a Good Policy for Music Streaming

A good policy for music streaming identifies the right "knobs" to turn, and to what extent they should be turned, in order to accommodate three goals: (i) avoid any pauses in playback, (ii) avoid needlessly using cellular bandwidth and instead use buffered content whenever possible, and (iii) avoid a significant reduction in battery life. Our study shows that the key to achieving these goals is making use of Tango's support for multi-level policy, i.e., using input from both user and app. Further, constraints are particularly useful for reducing cellular usage by protecting against "overeager" apps.

Data reductions on cellular. When evaluating our first two goals, `Unl` serves as our baseline, while `Rate` and `App` introduce user policy and multi-level policy, respectively. We can compare the buffer graphs for these configurations in Figure 5 with those for plain Android (Figure 4). These configurations all avoid any pauses, and their buffer decreases on WiFi (red) are less frequent and shorter, meaning that with this switching method data is buffering when plain Android is blocked on non-working WiFi. However, as Figure 6 shows, a consequence of moving off WiFi is more cellular usage—up to 6-7x compared to plain Android for this trace.

Better connectivity is in general a good thing, but it is contrary to our goal of reducing cellular usage. Applying `Rate` and `App` policies drastically reduces cellular usage while maintaining a pause-free playback. However, only `App` has the right combination of user-policy constraints and app-policy knowledge to reduce cellular usage to 30% of that of plain Android, which already has "artificially" low cellular usage due to clinging to WiFi.

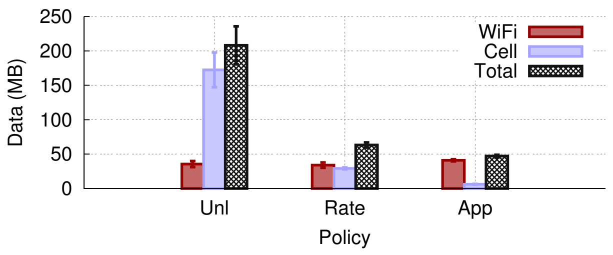


Figure 6: Network usage showing how Tango policies (`Rate` and `App`) can drastically reduce cellular usage compared to unlimited usage (`Unl`).

Battery usage. In light of our third goal for Tango policies, we sought to understand how cellular-reducing policies affect the smartphone's battery life. To evaluate this, we ran the music streaming app while our emulation trace looped until the battery ran out, recording the battery percentage at every second. Table 6 breaks down the rates of decline and total life of each policy. With 25% of battery left, the drain rate appears to speed up, possibly due to OS or the firmware attempting to avoid

	First 75% batt. drain (% / hr)	**Last 25% batt. drain (% / hr)**	**Batt. Life Life (hrs)**
`Unl`	-12.6	-19.2	7.25
`Rate`	-12.7	-19.5	7.21
`App`	-9.5	-13.9	9.70

Table 6: `Unl` and `Rate` keep cellular active and drain battery faster, while `App` is able to reduce the drain.

a complete drainage, so we segment the decline rates for the first 75% and the last 25%. We see that `Unl` and `Rate` experience similar battery life—about 7.2 hours—losing just 13% an hour for the first 75% and about 19% for the remainder. This suggests that battery life is dependent on the amount of time the cellular network is active, regardless of transmission rate.[4] On the other hand, `App` provides over 2 hours additional battery life, for a total of 9.7 hours. `App` lets the cellular radio transition to a low power idle state during times when the buffer is sufficiently full, saving power over `Unl` and `Rate`.

5.3 Case Study 2: Policy Across Apps

We now consider how Tango can provide fairness and prioritization *across* apps competing for resources.

5.3.1 App Fair Sharing

On mobile devices, apps that open up many flows can gain a higher share of bandwidth because of TCP's built-in fairness. For example, an app downloading several songs concurrently for later listening would drown out a single-flow video stream. Since users typically think in terms of apps rather than flows, app-level fair sharing can be a more natural fit for expressing a user's needs. This is achieved in Tango with a user policy setting a constraint on each app to be $1/N$ of the available bandwidth.

We implemented such a policy in a scenario with multiple networks available simultaneously—giving apps a choice in which to use—but enforcing fair sharing on each link. In our scenario, the user policy rate limits the HSPA network to 640 kbps after 30 seconds to discourage its use, but removing the constraint after 30 seconds of idleness. We have two apps: a multi-flow app (MFA) with five flows that always uses the 2 Mbps WiFi link, and a single-flow app (SFA) with an app policy to migrate to the network where it gets the best performance. When simultaneous interface usage is acceptable, Tango helps reduce costs and/or optimizes performance by scheduling flows more intelligently.

Figure 7 shows these results. For the first 30 seconds, SFA uses cellular as its measured speed is better than its WiFi constraint (~1 Mbps). When the user policy constrains cellular to 640 kbps, however, SFA migrates to WiFi, equally sharing the WiFi bandwidth with MFA,

[4]We believe the higher drain for `Rate` is mostly noise due to external factors, i.e., the load on the cell network.

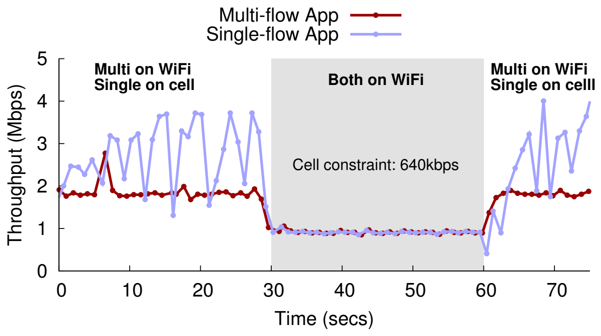


Figure 7: App-level fair sharing at link level, while app policy optimizing performance given constraints.

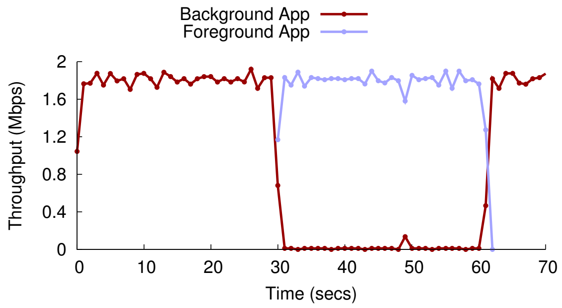


Figure 8: Providing priority dynamically to foreground traffic for better user experience.

despite the latter's 5 flows. Once the constraint on cellular is lifted, SFA moves back to cellular. Not only have we achieved fairness when sharing a link, but app policy enables the single app to achieve better performance by responding to the constraints.

5.3.2 Dynamic App Prioritization

In some cases, prioritizing certain app network usage is more desirable for a user than fairly sharing resources. For example, congested or slower cellular links may not have enough bandwidth for simultaneously syncing photographs with the cloud (background) *and* web browsing (foreground). With Tango, a user policy can dynamically prioritize an interface's bandwidth, demonstrated in Figure 8. At time 30s, the user opens an app in the foreground. The policy strictly prioritizes the foreground app, giving it the full link rate until the app closes at 60s.

5.3.3 Fairness, Priority, and App Hints

Tango allows app policies to send hints to the user policy, which provides a powerful way to improve prioritization across apps. In the previous example, the policy works on the assumption that the foreground app is always more important than the background app to the user. This is not always true, however, e.g., music streaming in the background. When the music app's playback buffer gets low, its network usage becomes important to prevent a user from experiencing playback pauses. This is a prime opportunity to use Tango's app hints.

To demonstrate this, we combine the per-app fairness and prioritization from the previous examples with app hints. We have two apps competing for a 550 kbps WiFi link: a web app that continually downloads `yahoo.com`'s frontpage (including any embedded or Javascript-initiated content) and our music streaming app. Figure 9 shows how this situation performs on today's smartphones. The page load times, as measured by Android's WebView, are low until the music app begins in the background. Once that happens, the page load times and variability increase, while the music indiscriminately adds to its playback buffer. The long-running music app is able to fill kernel queues, preventing the burstier web app from getting its fair share due to losses in TCP slow start. This lack of resource isolation makes using multiple apps on the phone a poor experience.

Tango can address this problem with its policy enforcement. We implemented a user policy that has two priority classes—high and normal—and fair shares network resources between apps of the same class. The web app always runs at high priority to provide a low-delay experience while browsing. The music app policy sends a hint to the controller that it wants high priority whenever its buffer falls below 20s; once the buffer is above 30s, the app sends a hint for normal priority, as its network needs are less urgent. The user policy uses these hints to set the priority constraints for the music app.

Figure 10 shows the result of running both apps with this policy. The white area is when the web app runs alone, lightly shaded areas have both apps are running at high priority, and darker shaded areas have the music app is running at normal priority. When the browser runs alone or the music app runs at normal priority, the page load times remain low, as the browser is strictly prioritized over the music app. Page load times only increase when the music app needs to replenish its buffer (and hence increases its priority). But the load time increase is more modest and less variable than in Figure 9, as the fair sharing between the two high-priority apps gives each their own queue, reducing losses in the browser's TCP slow start. Given the work-conserving nature of the HTB setup, the music app also improves its download rate when the web app stops at time 360. In short, Tango's hints allows apps to intelligently cooperate in order to improve overall user experience.

These exemplify just a few policy decisions that are useful *across* apps. Tango's constraint mechanism allows limits to be adjusted on a per-interface basis as well, enabling users to set different management strategies based on the network type. These techniques generalize: user policies can also deal with classes of apps and foreground/background status. Further, apps can optimize in the presence of constraints that restrict their network usage. Non-critical flows can be deferred or given the lowest priority within the app, so that they only consume network resources after other flows complete.

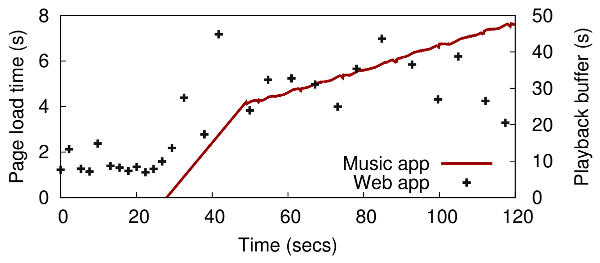


Figure 9: Today's phones insufficient resource isolation: background music reduces web performance.

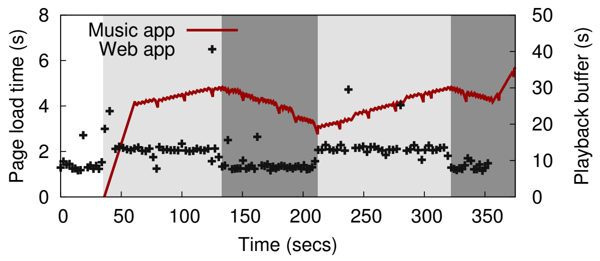


Figure 10: Providing dynamic priority between music and web. The music app hints at its need for the network to provide improved page load times when its buffer is healthy (dark gray) and minimal disruption when its buffer needs replenishing (light gray).

6 Related Work

The advent of mobile computing has led to the development of several approaches to incorporating "context" into programming for richer application models. JCAF [4] is a context framework for Java applications consisting of "entities" that both make up context and respond to changes in other entities. It is not tailored for mobile and does not focus on resource management, but rather getting entities to respond in the presence of other entities. Tango instead focuses on the mobile platform and managing resources which can be quite scarce. JCAF's somewhat open-ended nature would be make it difficult to handle things that Tango does, like the user and app policy split and our constraint model. CASS [6] is another context framework that uses nearby sensors and a remote server to create a context view to supply to apps. It abstracts away details to simplify policy writing (e.g., converting a temperature reading into a state list of "cold," "normal," and "hot"). Tango is instead completely local to the device, both in terms of sensing data and compiling the resultant state. Tango also leaves the level of abstraction up to policy writers by providing mostly raw metrics. CARISMA [5] is a context framework that attempts to resolve conflicts not only on-device but also amongst multiple devices using the same app. It uses utility functions and a sealed bid auction as its conflict resolution mechanism. Tango does not attempt to coordinate policies amongst multiple devices, but instead focuses on dealing with conflicts between policies on the same device. Further, our constraint mechanism helps deal with these conflicts proactively, rather than use utility functions, which are hard to define for the potentially large policy space.

Several earlier projects focus on selecting between multiple wireless networks. Ormond et al. [13] uses a utility-based approach to minimize costs when uploading files by choosing between multiple networks with varying costs and bandwidth, subject to time constraints. Wilson et al. [20] uses a fuzzy-logic inference engine that takes input from both user and apps and, based on predefined QoS metrics and rules, decides on the preferred network. Ylitalo et al. [24] presents an interface selection framework where flows can be moved between several networks. It requires some changes to the socket API and uses a rule-based approach for selection. Tango's flexible programmatic model supports these rule- and utility-based approaches, as well as considerably richer policies. Further, Tango's app policy and hints allow for broader app input, yet still avoids OS or socket API modifications. Finally, beyond network choice, Tango addresses deeper control of the network by exposing management control of traffic queues.

Other prior work focuses specifically on reducing cellular usage through WiFi offloading. One body of research has tried to generalize application-specific prefetching strategies by providing middleware that *batches* data for download during periods of WiFi connectivity. Lee et al. [9] describes a simulated batching strategy that delays transfers in anticipation of future WiFi connectivity. Wiffler [3] employs another batching strategy that adds prediction of WiFi throughput to determine whether transfers would complete within a WiFi connectivity window. This requires prior knowledge of data sizes and accurate WiFi prediction. IMP [8] also performs batching on WiFi, but may also (pre)fetch on cellular if allowed by budget constraints that take into account battery and data usage. BreadCrumbs [10] tracks user mobility and network conditions to forecast network connectivity, and the authors discuss its use to inform prefetching and batching strategies. SALSA [16] employs similar forecasting, using an energy-delay trade-off algorithm to select the energy-minimizing link for a data transfer. These techniques are complementary to Tango's general framework, and similar batching strategies may be adopted by specific delay-tolerant apps running on Tango to optimize their resource usage.

In contrast to this prior work, Tango can continue data transfers despite changing connectivity, relying on ECCP [2] for migrating TCP connections. Although no batching is done, the bulk of data transfers may be moved to WiFi by rate limiting cellular links, in anticipation of future WiFi connectivity. This ensures transparent support for interactive or latency-sensitive applications (e.g., live video streaming), even if initiated while on cellular. Prefetching and excessive buffering on cellular is also avoided, which could otherwise deplete a user's data cap or battery resources. Even so, the prior work on forecasting and link estimation could help inform Tango polices for more accurate rate limiting and migration decisions.

Recent work [14, 23, 2] has explored seamless use of heterogeneous networks using migration techniques, based on MPTCP [21] and OpenVSwitch. Tango could adopt those or alternative migration techniques, including Mobile IP [15], HIP [11], LISP [7], or TCP Migrate [17]. Unlike such work, we use flow migration as just one of many techniques that, in combination, enable interesting control plane and policy control to better utilize available networks, while simultaneously accounting for device and user needs.

7 Conclusion

The unique challenges posed by mobile devices necessitates re-examining how they provide resource management. This paper argues for the adoption of a programmatic policy model, rather than ad-hoc and static configuration settings, in order to better support the important, dynamic, and diverse interests of mobile users and apps. Our resulting system, Tango, tackles some challenging architecture and interface problems in how to enable these parties to align their interests and optimize their behavior. We examine several scenarios where Tango could greatly improve mobile computing today. Even so, our case studies provide only a small sample from the vast space of tailored policy options that we believe are possible using such a framework.

References

[1] A. Aleryd. How Sony's Battery STAMINA Mode works. `http://developer.sonymobile.com/2013/04/03/how-sonys-battery-stamina-mode-works/`, Apr. 2014.

[2] M. Arye, E. Nordström, R. Kiefer, J. Rexford, and M. J. Freedman. A Formally-Verified Migration Protocol For Mobile, Multi-Homed Hosts. In *ICNP*, Oct. 2012.

[3] A. Balasubramanian, R. Mahajan, and A. Venkataramani. Augmenting Mobile 3G Using WiFi. In *MobiSys*, June 2010.

[4] J. E. Bardram. The Java Context Awareness Framework (JCAF) - A service infrastructure and programming framework for context-aware applications. In *Pervasive Computing*, May 2005.

[5] L. Capra, W. Emmerich, and C. Mascolo. CARISMA: Context-aware reflective middleware

system for mobile applications. In *IEEE Transactions on Software Engineering*, Oct. 2003.

[6] P. Fahy and S. Clarke. CASS - A middleware for mobile context-aware applications. In *Workshop on Context Awareness at MobiSys*, June 2004.

[7] D. Farinacci, V. Fuller, D. Meyer, and D. Lewis. Locator/ID separation protocol (LISP), draft-ietf-lisp-22, Feb. 2012.

[8] B. Higgins, J. Flinn, T. Giuli, B. Noble, C. Peplin, and D. Watson. Informed Mobile Prefetching. In *MobiSys*, June 2012.

[9] K. Lee, I. Rhee, J. Lee, S. Chong, and Y. Yi. Mobile Data Offloading: How Much Can WiFi Deliver? In *CoNEXT*, Nov. 2010.

[10] A. J. Nicholson and B. D. Noble. BreadCrumbs: Forecasting mobile connectivity. In *MOBICOM*, Sept. 2008.

[11] P. Nikander, A. Gurtov, and T. R. Henderson. Host Identity Protocol (HIP): Connectivity, Mobility, Multi-Homing, Security, and Privacy over IPv4 and IPv6 Networks. *IEEE Comm. Surveys*, 12(2), Apr. 2010.

[12] E. Nordström, D. Shue, P. Gopalan, R. Kiefer, M. Arye, S. Ko, J. Rexford, and M. J. Freedman. Serval: An End-Host Stack for Service-Centric Networking. In *NSDI*, Apr. 2012.

[13] O. Ormond, J. Murphy, and G.-M. Muntean. Utility-based Intelligent Network Selection in Beyond 3G Systems. In *IEEE ICC*, June 2006.

[14] C. Paasch, G. Detal, F. Duchene, C. Raiciu, and O. Bonaventure. Exploring Mobile/WiFi Handover with Multipath TCP. In *CellNet*, Aug. 2012.

[15] C. Perkins. IP Mobility Support for IPv4, Revised (RFC 5944), Nov. 2010.

[16] M.-R. Ra, J. Paek, A. B. Sharma, R. Govindan, M. H. Krieger, and M. J. Neely. Energy-Delay Tradeoffs in Smartphone Applications. In *MobiSys*, June 2010.

[17] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *MOBICOM*, Aug. 2000.

[18] I. van Beijnum. Multipath TCP lets Siri seamlessly switch between Wi-Fi and 3G/LTE. `http://arstechnica.com/apple/2013/09/multipath-tcp-lets-siri-seamlessly-switch-between-wi-fi-and-3glte/`, Sept. 2013.

[19] Web10G. `http://web10g.org/`, Mar. 2013.

[20] A. L. Wilson, A. Lenaghan, and R. Malyan. Optimising Wireless Access Network Selection to Maintain QoS in Heterogeneous Wireless Environments. In *WPMC*, Sept. 2005.

[21] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, implementation and evaluation of congestion control for multipath TCP. In *NSDI*, Mar. 2011.

[22] Q. Xu, J. Erman, A. Gerber, Z. M. Mao, J. Pang, and S. Venkataraman. Identifying Diverse Usage Behaviors of Smartphone Apps. In *IMC*, Nov. 2011.

[23] K.-K. Yap, T.-Y. Huang, M. Kobayashi, Y. Yiakoumis, N. McKeown, S. Katti, and G. Parulkar. Making Use of All the Networks Around Us: A Case Study on Android. In *CellNet*, Aug. 2012.

[24] J. Ylitalo, T. Jokikyyny, T. Kauppinen, A. J. Tuominen, and J. Laine. Dynamic network interface selection in multihomed mobile hosts. In *HICSS*, Jan. 2003.