Socialize Spontaneously with Mobile Applications

Zimu Liu, Yuan Feng, Baochun Li
Department of Electrical and Computer Engineering
University of Toronto
{*zimu, yfeng, bli*}*@eecg.toronto.edu*

***Abstract*—With the proliferation of mobile devices in both smartphone and tablet form factors, it is intuitive and natural for users to socially interact with their collaborators or competitors in multi-party conferencing, productivity, or gaming applications. In this paper, we make a case that such social interactions should be much more *spontaneous* to users in these applications. We design and implement a new system framework, *Reflex*, to provide the required system support to achieve spontaneous social interaction with other users in the same mobile application, be they in the same living room or around the world. *Reflex* features a simple and intuitive application programming interface (API), and uses cloud computing services from Google App Engine to offer the scalability and performance required to support spontaneous social networking at a large scale. *Reflex* is able to transparently switch to local interactions over Bluetooth or Wi-Fi interfaces, available on mobile devices, whenever possible. In order to evaluate *Reflex* in the iOS platform, we developed a real-world music composition application, called *MusicScore*, from scratch on the iPad, which takes advantage of *Reflex* to let music composers collaborate in real time.**

I. Introduction

It is no longer a disputable fact that mobile devices, in both tablet and smartphone form factors, have become mainstream computing devices for the general population to work and interact more effectively with their colleagues, clients, and friends in social circles. The current generation of social networks has changed the way people socialize online, making words such as "liking," "friending," "status," or "following" commonplace. But where are future social experiences *trending*? We believe that the answer lies in two words: *apps* and *spontaneity*.

In-app social experiences. we argue that social interactions will soon migrate from traditional online social websites to mobile devices, in dedicated mobile applications, or more simply, "apps." The need for social interaction with other users has already appeared in mobile game applications that require multi-party gameplay, both collaboratively and competitively. Yet, such a need will soon evolve to a wide range of mainstream mobile applications designed for productivity, media creation, and media consumption. Imagine the level of satisfaction if a photographer can interact with his team members live while editing photos; if a designer can show an illustration sketch to her clients in a remote location; or if fans anywhere in the world can watch a live concert event together while interactive comments can be exchanged with a variety of media alternatives, such as text and audio streams.

Spontaneity. We are convinced that a wide variety of mobile applications, including multi-party games, need to allow users to interact socially with their partners *spontaneously*. Spontaneity implies that users may not be "friends," and they do not need to create social credentials or log in with such credentials before social interactions may occur. Yet, spontaneous social interaction progresses in real time (as in online chat), rather than asynchronously (as in Twitter). Thanks to dedicated mobile applications, such spontaneous social interactions involve much richer experiences than status messaging or text chat, typical in online social networks today.

At first glance, features provided by such a framework seem to be widely available: after all, an instant messaging (IM) or social networking platform, such as MSN Messenger, Skype, or Facebook, allows users to spontaneously interact with one another in a social circle in real time, while supporting text, audio, and even video streams. One can only imagine the surprise and disappointment when our extensive research discovered no available system frameworks off-the-shelf to support in-app spontaneous social interaction, as we try to implement such social interaction features in a new real-world music composition application on the iOS platform.

Existing IM or social networking platforms do not offer a suitable solution for a number of reasons: *First,* they are proprietary and custom-designed for purposes of spontaneous messaging, with no effort devoted to reusability. It would be difficult to use them directly in mobile applications. *Second,* they require users to create new accounts or to login with existing ones in a proprietary system, and to remember their login passwords. *Third,* all messages are strictly relayed by IM or social network servers, even if users are at the same location within the range of Bluetooth or Wi-Fi. Last but not the least, they may not be able to handle bursty exchanges of *application-specific states* — such as avatar updates in an action game — in a short timespan. In conclusion, when users in a game or media creation application need to spontaneously interact and exchange application-specific states in real time, we believe that it is not feasible to use proprietary IM or social networking solutions.

In this paper, we design and implement *Reflex*, a new system framework we have developed on the iOS platform. *Reflex* takes full responsibility for maintaining online presence status updates, and for managing application-specific live data broadcast sessions among users, be they in the same living room or across the world. *Reflex* takes advantage of existing cloud services such as Google App Engine, yet transparently switches to local interactions using Bluetooth or Wi-Fi interfaces whenever possible. It is designed with *reusability* as a first-class goal, and with carefully chosen design patterns to maximize its

flexibility and to minimize tight coupling across components. It supports the relay of data streams to other users via a number of *reflectors* in the cloud. In order to evaluate *Reflex*, we developed a real-world music composition application, called *MusicScore* as shown in Fig. 1, from scratch on the iPad. We show an extensive array of experimental results using *Reflex* in *MusicScore*, assisted by existing cloud services in Google App Engine and PlanetLab.

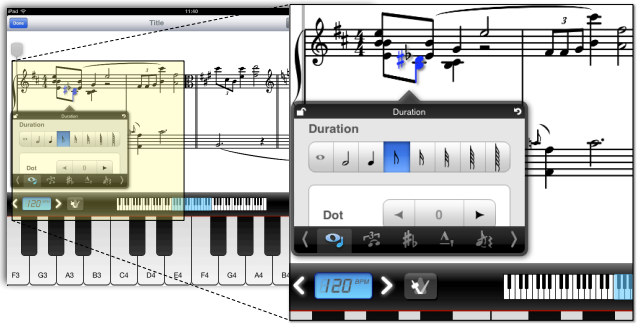

Fig. 1. A screenshot of running *MusicScore* on the iPad.

The remainder of this paper is organized as follows. In Sec. II, we present our objectives when designing our system framework. In Sec. III, we show how *Reflex* spontaneous presence is designed and implemented in the cloud to achieve our objectives. In Sec. IV, we reveal details of *Reflex* framework for mobile applications. In Sec. V, we present results from our real-world experiments using *Reflex* and *MusicScore*. We conclude the paper with a discussion of related work and final remarks in Sec. VI and VII, respectively.

II. Reflecting on the Reflex Design

To architect the best possible system framework design for mobile social interaction, we believe that *Reflex* should focus on features that off-the-shelf IM and social networking solutions are not able to readily provide to mobile applications. With respect to the set of features, we believe that *Reflex* should seamlessly support *spontaneous presence* that is custom-tailored to mobile users, as well as *live streaming sessions* of application-specific states. With respect to performance, our design and implementation of *Reflex* focus on scalability and energy efficiency. With respect to the Application Programming Interface (API), *Reflex* should take full responsibility of mundane aspects of mobile social interaction, yet providing sufficient flexibility for alternative mechanisms specific to a mobile application to be deployed. With respect to implementation, we have chosen to implement *Reflex* in the iOS platform, widely adopted by commercial mobile application developers.

Spontaneous Presence

Spontaneity. The thesis of the *Reflex* framework is to support *spontaneous* social interaction. With existing spontaneous presence systems, such as Apple's *Game Center*, Google Talk, and Facebook, mobile users will need to explicitly register using their email addresses, and login with a password. Such curated online presence is adequate when security and privacy are primary concerns, but is too cumbersome for spontaneous and live social interactions within mobile applications. We believe that *Reflex* should be designed to function well without explicit user registration: a mobile user may simply use an anonymous user profile, such as a nickname, to join a socially interactive session and start socialize with others spontaneously. This way, the threshold to interact with others is as low as it gets. If the need arises, a user can certainly update her profile with her social network profiles (*e.g.*, from Facebook or Twitter) at any time.

Persistence. Routinely, a running application in a mobile device may be forcefully terminated due to an incoming call or a network failure (*e.g.*, when a user roams to a location with weak signal strength). As such, spontaneous presence in *Reflex* should provide superior support to the persistence of states in live social interactive sessions, allowing a user to restore her previous status after a brief "offline" period.

Live Social Interactive Sessions

To relieve developers from the burden of network programming on mobile devices, *Reflex* should be designed to encapsulate mundane details of low-level network programming, providing a simple high-level API that supports live social interactive sessions. It should seamlessly integrate available network interfaces, including 3G cellular, Bluetooth, and Wi-Fi, and carefully monitor and control their usage to achieve the best possible performance and energy efficiency.

Performance

As a system framework serving users with mobile devices, it is critically important for *Reflex* to deliver a smooth online interactive experience with live streaming sessions of application-specific states. With the potential of large flash crowds attending live social events within a mobile application, *Reflex* needs to be *robust* and *scalable* to allow a large audience in live sessions. In addition, *Reflex* needs to be *energy efficient*, and conserves battery power in mobile devices as much as possible, by making adequate selections of network interfaces when data are to be transmitted.

Reusable and Flexible API

As a system framework, *Reflex* must be highly reusable and flexible enough to support a wide variety of applications with social interaction needs. Without distracting developers from implementing the core functionality of their own applications, *Reflex* should hide internal details that support spontaneous presence and live interactive sessions. *Reflex* should function well *off-the-shelf*, with a simple interface. For applications requiring custom-tailored mechanisms, it must be able to incorporate them as *delegates* using carefully chosen design patterns.

III. The Reflex Cloud for Spontaneous Socializing

To achieve our objectives towards the best possible design of the *Reflex* framework, we present highlights in our design from two different perspectives: *in the cloud* and *on mobile*

devices. In the cloud, we build a platform using existing cloud services to support spontaneous presence and live interactive sessions. On mobile devices, we focus on transparent switching among different network interfaces, as well as a reusable and flexible API to developers. In this section, we discuss how *Reflex* uses support from the cloud to provide core services for spontaneous social interactions, with a focus on its design issues and implementation details.

A. Designing Spontaneous Presence with Cloud Support

As the principal goal of the *Reflex* is to support spontaneous social interaction among mobile users, the *Reflex* service in the cloud should allow mobile applications to easily join the spontaneous presence system, query other online users, and initiate spontaneous online collaboration or competition, all without the need of creating user accounts. Conceptually, a mobile application built on the *Reflex* framework communicates with front-end servers using its unique mobile device identifier, and obtains a customizable online profile dynamically created for this device. By personalizing the profile (such as specifying a nickname, location, gaming skills, or preferences), online users with similar characteristics are able to find each other easily to start a live interactive session, *i.e.*, explicit "friending" is unnecessary in *Reflex*. The spontaneous presence service keeps all online profiles well organized in the database for searching, and coordinates the creation of social interactive sessions among multiple users with common interests. In Fig. 2, we present an illustration of how the *Reflex* spontaneous presence is designed.

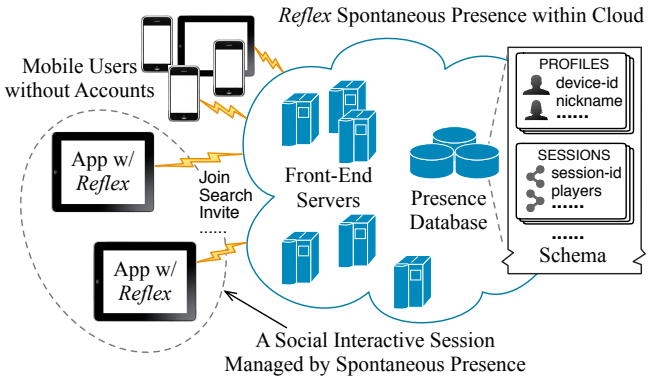

Fig. 2. The design of *Reflex* spontaneous presence.

To design a spontaneous presence service supporting a growing population of mobile users, our primary objective is to scale up to a large crowd of spontaneous users joining live sessions. To achieve this goal, the traditional approach is to build a distributed registration service from scratch, and then deploy it over a group of physical or virtual servers in the Internet. Such an approach would lead to an insurmountable level of complexity as the number of online users scales up, in order to maintain high levels of data consistency and permanence.

In contrast, we believe it would be better to take full advantage of *Platform as a Service* (PaaS) in the cloud [5], provided by cloud service providers such as Google App Engine (GAE) [2]. PaaS delivers a complete solution stack (including programming/scripting runtime, generic database support, and communication interfaces) and facilitates the development and deployment of network-centered services, without the cost and complexity of purchasing and managing the underlying hardware and software layers. Specifically, we choose GAE as the cloud backend to host *Reflex*. With the support of GAE, the *Reflex* spontaneous presence cloud is designed to simultaneously handle HTTP requests, process various queries, and deliver push notifications. GAE allows the *Reflex* cloud platform to run concurrently and independently across distributed servers, and provides it with a persistent, yet highly scalable, distributed database interface, called *datastore*, for data storage and retrieval. These features match well with our requirements for a scalable spontaneous presence service.

Design Choices for Presence Communication

GAE provides two core services for live communication with end users: HTTP and XMPP. Intuitively, XMPP seems to be a good fit for spontaneous presence. However, it turns out that XMPP in GAE can only work in client mode, *i.e.*, we have to setup our own XMPP servers to relay message streams between *Reflex* and the GAE backend, which to some extent defeats the purpose of using PaaS. In contrast, we do not need to run additional servers if the self-contained HTTP service is used. In *Reflex*, to allow HTTP to transport our customized requests or queries, we adopt JSON-RPC over HTTP as the communication tunnel for spontaneous presence. In a nutshell, all spontaneous presence requests are sent over HTTP POST with JSON-encoded messages, and the corresponding results are returned by our GAE service handler as HTTP responses. Fig. 3 illustrates how a device joins into the *Reflex* spontaneous presence service via JSON-RPC. To protect the privacy of user profiles and improve system security, SSL is enforced across all HTTP connections.

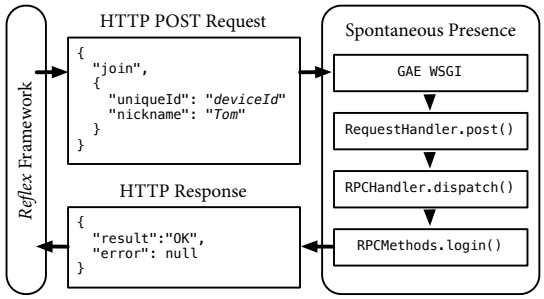

Fig. 3. A user joins *Reflex* spontaneous presence using JSON-RPC.

However, we notice that due to the nature of HTTP, all information from the cloud has to be passively carried by responses. What should we do when the cloud actively generates a notification for a user? To make spontaneous presence capable of sending messages to mobile users, *Reflex* provides both a passive polling mechanism, and an active push notification service. For periodic and non-essential information updates

from the cloud, periodic polling will be initiated from users as regular JSON-RPC queries. Urgent notifications, such as a collaboration invitation, can be send through GAE's one-way communication tunnel, called a *channel*. As illustrated in Fig. 4, whenever a user connects to the *Reflex* spontaneous presence cloud, it obtains a secret channel token from the cloud, and listens on the incoming push channel identified by the token, using a JavaScript-based *channel* client embedded in the *Reflex* framework. Although the embedded JavaScript client in *Reflex* is not as efficient as native Objective-C code, it is sufficient to accommodate short notification messages for regular operations.

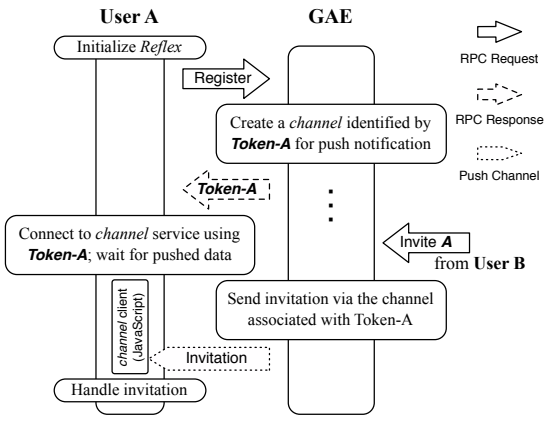

Fig. 4. *Reflex* actively sends a push notification through a *channel*.

B. Implementation Challenges in GAE

Although GAE provides us a powerful yet convenient PaaS cloud platform, there is always a flip side of the coin. To be a more scalable infrastructure, GAE enforces certain limitations to its clients. Due to GAE's unique characteristics and various constraints, implementing a reliable and scalable spontaneous presence is not without its challenges.

Choice of the Database Structure

The *datastore* facility provided in GAE is based on BigTable [6], a sparse distributed multi-dimensional sorted map. In *datastore*, each data object is stored as an entity with multiple properties, which can be manipulated or queried using a simplified *datastore* interface. Different from relational databases, it lacks support of foreign keys and complex filtering. In particular, *datastore* does not support inequality filters on more than one entity property per query. In this sense, for social spontaneous presence, we have to carefully design table structures to support complicated social search. For example, in GAE it would be impossible to find friends within a given geographical region by conducting a query with two inequality filters on both longitude and latitude. To overcome this issue, whenever a user reports her location, we not only store her geographical coordinates, but also store the corresponding geohash value [1]; and the database index is build over the geohash property, rather than over actual coordinates.

Since social networks may gradually evolve over time, it would be best to maximize the flexibility of underlying database structures for future feature expansions. In the *Reflex*'s underlying database structure, we adopt the *Expando* data model in GAE, whose properties are determined dynamically. This model allows both fixed properties to be used as Reflex's essential data fields (*e.g.*, a user's device ID and nickname), and dynamic properties to be customized by different developers or users with various preferences. With the help of such a dynamic model, the complexity of adding additional social features can be greatly simplified.

Minimizing the Processing Time

For the sake of fairness among applications, GAE imposes a 30-second limit on responding to a request. To meet such a processing deadline in *Reflex*, we decide to move a part of jobs from foreground RPC handlers to GAE's *task queues* in the background, which allows much longer processing times (around 10 minutes). For example, once a user has uploaded a profile image, producing thumbnails in different sizes can be executed in the background, as shown in Fig. 5(a). Such a foreground/background job partition also leads to smooth and responsive interactions between the cloud and the mobile application, which is important for a smooth user experience.

We also notice that a substantial portion of the time is consumed on processing queries to the *datastore*, which can be accelerated by using an in-memory data cache, called *memcache*. For frequently accessed data, *Reflex* caches them after the first *datastore* query, and subsequent queries can be directly retrieved from *memcache*, as shown in Fig. 5(b). Additionally, we take advantage of the *cron* service in GAE to process heavy workloads that can be executed periodically, by scheduling them as *scheduled tasks*.

(a) *task queues*: producing thumbnails in the background

```
def uploadProfileImage(...):
  self.updateProfileImage(userId, image)
  taskqueue.add(url='/background/generateThumbnail?target=
      userId=%s&sizes=48,128' % userId);
```

(b) *memcache*: caching a user's profile

```
def getUserProfile(...):
  profile = memcache.get(userId)
  if profile is None:
    profile = self.queryUserProfile(userId)
    memcache.add(userId, profile)
  return profile
```

Fig. 5. Performance tuning with *task queues* and *memcache*.

C. The Reflector Cloud

After users establish a live multi-party interactive session using the spontaneous presence service, it is up to *Reflex* to relay data traffic in such a session. Intuitively, forming a complete mesh topology among users in the session may not be scalable, bandwidth-efficient, or energy-efficient. As an alternative solution, bandwidth usage within a complete mesh can be reduced by nominating a device as a session "host,"

relaying streams for others. However, power consumption on such a host will inevitably increase, which may not be fair to the host. In addition, direct device-to-device connections may not even be feasible in most cases, as Network Address Translation (NAT) is extensively used on mobile devices.

In *Reflex*, we resort to the deployment of *reflectors* in a *reflector cloud*, which helps with the aggregation and relay of data streams in live interactive sessions. Devices connect to a common set of nearby reflectors in the cloud, and data are broadcast to receivers in the session via two-hop relay paths over these reflectors. To achieve timely delivery from users around the world, it would be best to distribute reflectors to different geographical locations in the cloud. For this reason, we choose to use Infrastructure-as-a-Service (IaaS) cloud service providers, which are based on Virtual Machine instances (VMs) across the world with excellent connectivity in the Internet, to host and implement our reflectors. To implement an efficient reflector in a VM-based cloud platform, we need to choose the most suitable design between two alternatives: (1) a thread-based server that handles simultaneous connections in separated threads, with a small amount of memory overhead; and (2) an event-driven server, handling requests asynchronously in very few threads. *Reflex* adopts the latter alternative to avoid the scheduling overhead of multi-threading. We implement reflectors using Python and the *Twisted* asynchronous networking engine with full event-driven networking support. Each reflector instance in *Reflex* runs in a VM in the cloud, and registers itself in the spontaneous presence service via the same JSON-RPC interface.

IV. The Reflex Framework in Mobile Applications

From the application's point of view, *Reflex* provides a unified API hiding all underlying network plumbing that manages spontaneous presence and socially interactive sessions. Internally, application-specific data in interactive sessions are dispatched by *Reflex*, either to *reflectors* in the cloud, or to locally connected sessions over Bluetooth or Wi-Fi interfaces. Social profiles and session information in *Reflex* are under the control of dedicated "managers." Application-specific policies can be "plugged in" via *delegates*. Fig. 6 presents an architectural overview of *Reflex*.

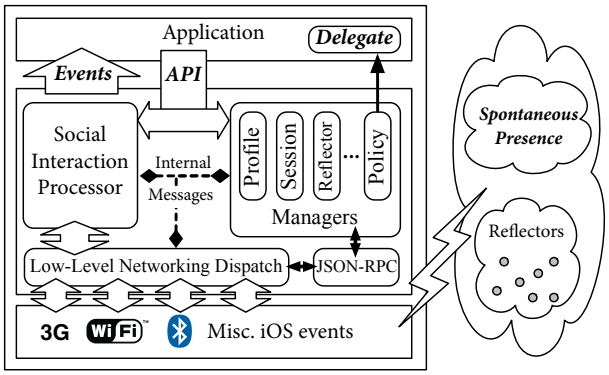

Fig. 6. An architectural overview of *Reflex* framework for mobile applications.

TABLE I
Characteristics of network interfaces on mobile devices [4].

	Bandwidth	Active (J/MB)	Idle (W)	Scan (W)	Range (m)
Cellular	100+ kbps	100	−*	−*	500
Wi-Fi	11+ Mbps	5	0.77	1.29	100
Bluetooth	700 kbps	0.1	0.01	0.12	10

Note: the cellular interface is usually on.

A. Local Interactions

In Table I, we present the strong motivation for *Reflex* to transparently switch to Bluetooth or Wi-Fi network interfaces whenever possible, with respect to both performance and power efficiency. As shown in Fig. 7, when some users in an online session are at the same location (*e.g.*, user A and B), there may be an opportunity to connect them using Bluetooth or Wi-Fi networks in an *ad hoc* manner (shown as solid lines). The *Reflex* framework transparently discovers such a local interaction opportunity over Bluetooth or Wi-Fi, so that communication among nearby devices are no longer relayed by reflectors in the cloud.

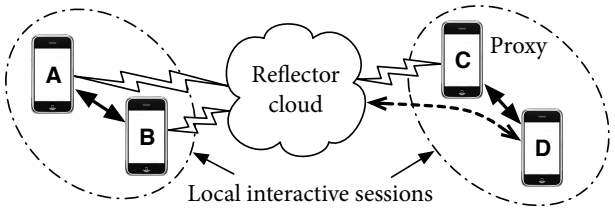

Fig. 7. Local interaction vs. the use of reflectors in a 4-party live session.

This feature of the framework also makes it possible when a user without Internet access (*e.g.*, user D in Fig. 7) wishes to join the session. D can connect to a nearby device with access to reflectors in the cloud (*e.g.*, user C), and with C's permission, C may act as a *proxy* that sends and receives data on behalf of D (shown as a dotted line). Again, the *Reflex* framework is responsible for the design and implementation of such a proxy, completely transparent to applications using the framework.

The iOS platform provides an excellent API that allows two or more nearby devices to create and manage an *ad hoc* Bluetooth or local wireless network (Wi-Fi). Given a specific identifier, devices are able to discover one another to form a local interactive session. One of the modes for such discovery is the *peer* mode, in which a device acts as both a server and a client, and is able to accept incoming connections and establish outgoing ones at the same time. Once devices are interconnected within the session, they are granted to exchange data, either reliably as ordered data streams, or unreliably as datagrams. A packet can either be broadcast to all local participants via `[sendDataToAllPeers:withDataMode:error:]`, or be sent to an arbitrary subset of devices, identified by their `peerId`, using `[sendData:toPeers:withDataMode:error:]`.

To explore opportunities of creating local interactive ses-

sions, after a device has entered an online multi-party session, the *Reflex* framework will enter the discovery period using the peer mode for a short period of time. Such a discovery period will last no more than 30 seconds to conserve energy. The *Reflex* framework then establishes an ad hoc network among discovered devices and refrains from further advertising or searching. Accordingly, all data in the interactive session are addressed by *Reflex* to nearby devices directly via a Bluetooth or Wi-Fi network interface, rather than via reflectors in the cloud.

B. A Reusable and Flexible API

Aiming to provide a powerful but developer-friendly system framework for a wide range of mobile applications featuring social interaction, we have designed the programming interface carefully to make *Reflex* both reusable and flexible.

First, as a framework specifically designed for mobile devices in the iOS platform, *Reflex* provides an entirely *event-driven* and asynchronous programming interface. Following the *actions* design pattern, *Reflex* allows an application to register its own application controller as an event handler, as illustrated in Fig. 8. When *Reflex* has finished processing an internal event, the corresponding application-specific event handler will be invoked asynchronously, with results as parameters in the callback. Such a design enables communication between applications and the *Reflex* framework without tight coupling.

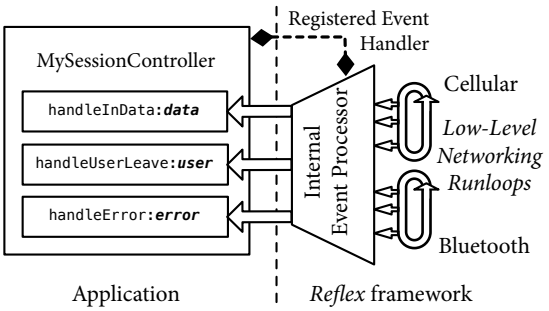

Fig. 8. The event-driven programming interface in *Reflex*.

Second, applications may wish to design their own algorithms to be used by *Reflex*, such as favouring a subset of reflectors in the cloud based on application-specific preferences. To make this possible, *Reflex* is able to load application-specific delegates in a loosely coupled manner. Designed with the *delegate* pattern, as shown in Fig. 9, when an application wishes to make decisions with its own algorithms, it simply implements an instance conforming to `ReflexPolicyDelegateProtocol`. It then registers this instance as a delegate of the *Reflex* framework. Whenever a decision needs to be made, *Reflex* will invoke a corresponding method in its delegate provided by the application, and use returned results to refine its final decision. The use of delegates in *Reflex* adds a substantial degree of flexibility, while minimizing its coupling with the application.

Finally, we also focus on making the design of *Reflex* itself as loosely coupled as possible, so that it is easy to expand and

`ReflexPolicyDelegateProtocol.h` given by *Reflex*

```
@protocol  ReflexPolicyDelegateProtocol
- (NSSet *)filterParticipantCandidates:(NSSet *)candidates
                             inContext:(NSDictionary *)ctx;
// other delegate method declarations
@end
```

`MyAppPolicy` instance provided by a developer

```
@interface MyAppPolicy : NSObject<
    ReflexPolicyDelegateProtocol> {
}
@end

@implementation MyAppPolicy
- (id)init {
  if (self = [super init]) {
    [[ReflexPolicyManager sharedManager] setDelegate:self];
  }
  return self;
}

- (NSSet *)filterParticipantCandidates:(NSSet *)candidates
                             inContext:(NSDictionary *)ctx {
  // app-specific participant candidate selection algorithm
  return aSetOfSuitableParticipants;
}
@end
```

A segment of code inside *Reflex*

```
id delegate = ReflexPolicyManager.delegate;
if (delegate != nil) {
  participants = [delegate filterParticipantCandidates:
      candidates
                                      inContext:ctx];
}
```

Fig. 9. Using an application delegate to select game participants.

to maintain. With liberal uses of decoupling design patterns such as actions, delegates and notifications, each component in *Reflex* has little or no knowledge of instances of other components. This makes it feasible to improve — or even replace — a component with relative ease. Taking the notification mechanism as an example, there is no strong coupling among components associated with a named notification (*e.g.*, `"ReflexPlayerOffNotification"`). Either the notification broadcaster or observer can be easily modified, as long as a globally unique notification name is defined in *Reflex*. For globally shared resources, corresponding *managers* are implemented with the *singleton* pattern.

V. Experimental Results

We believe that the best way to evaluate the *Reflex* framework is to implement it, along with a real-world mobile application that takes advantage of *Reflex* to enable spontaneous social interaction. Towards this objective, we have implemented *MusicScore*, a professional-grade multi-touch application for music composition (shown in Fig. 10). We developed *MusicScore* with about $59,000$ lines of code (LOC) in Objective-C. *MusicScore* takes full advantage of our *Reflex* implementation to allow composers to enjoy a live collaborative session, and students to benefit from a live educational experience, all without any knowledge of the architectural design choices in *Reflex*.

Beyond *MusicScore*, we have implemented the *Reflex* framework entirely from scratch, from mobile devices to the cloud

service, with about 3000 LOC. *MusicScore* produces actual multi-touch interaction workload that is used to evaluate the performance of *Reflex*. In order to collect run-time traces, a compact logging module has been implemented to anonymously record performance metrics as multi-touch interactions are streamed. To emulate a large number of concurrent online users and synthesize multi-touch streams, we implement a multi-touch trace playback module to replay user-interaction traces we captured in *MusicScore*. In this section, we present a performance evaluation of the *Reflex* framework.

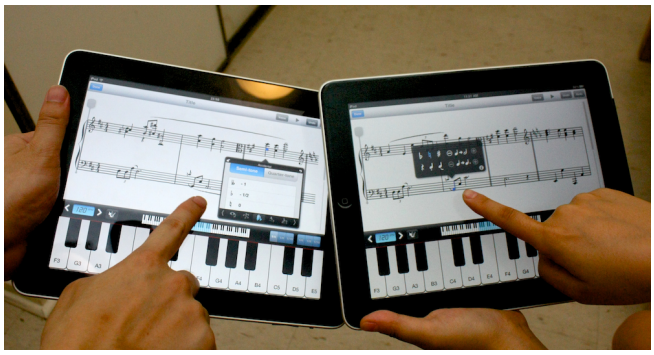

Fig. 10. *MusicScore* in action: two users are collaboratively composing a musical piece over local interaction support in the *Reflex* framework.

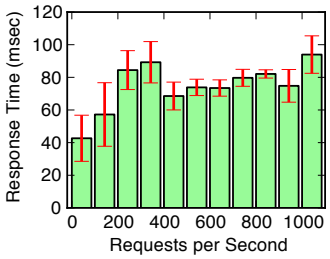

Fig. 11. Response time of the *Reflex* spontaneous presence service at different request rates.

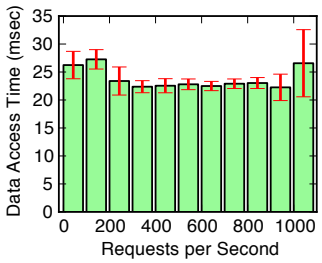

Fig. 12. Data access time consumed internally in the cloud at different request rates.

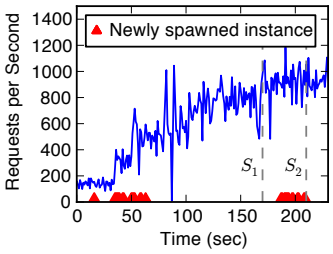

Fig. 13. GAE instance spawning when a flash crowd occurs.

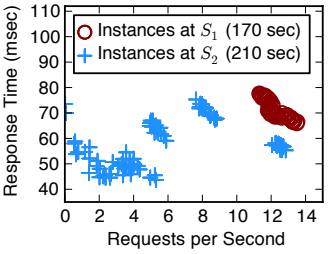

Fig. 14. Per-instance performance comparison between two snapshots in a flash crowd.

A. *Reflex* Spontaneous Presence on GAE

Our first experiment is designed to evaluate the spontaneous presence cloud, which serves as the "portal" of *Reflex*. We begin with an investigation of the performance and scalability of spontaneous presence at different request rates. As shown in Fig. 11, although the response time increases from 44 to 91 msec when the request rate increases from 50 to 300 requests per second, the response time is almost maintained around the same level when the request rate is high, and is no more than 100 msec. This can be attributed to both the excellent performance of GAE and our fine-grained performance tuning in the *Reflex* implementation, described in Sec. III.

Furthermore, we would like to study how well the underlying GAE infrastructure performs in *Reflex*. By taking a closer look at the data access time consumed by GAE's internal API, such as distributed *datastore* and *memcache*, in Fig. 12, we discover that there are no significant changes in the internal data access time when the request rate increases. These results further confirm that our choice of the Google App Engine for the spontaneous presence cloud is appropriate when it comes to scalability.

To conduct a more thorough examination on its scalability, we push the spontaneous presence cloud to its limits, when a flash crowd happens within a few minutes. By synthesizing a large volume of mobile devices, we closely monitor the resulting JSON-RPC request rate and the number of serving instances in GAE. As shown in Fig. 13, to guarantee the quality of service as the request rate increases, the GAE dynamically spawns more instances during $25-70$ sec and $175-205$ sec. Consistent with results in Fig. 11, the response time is less than 100 msec. Further, we examine two typical snapshots in our flash crowd traces, one before the second batch of spawning (S_1) corresponding to a sharp raise of requests starting from 163 sec and the other after the spawning (S_2). In Fig. 14, we observe that at S_1 before spawning, almost all instances are heavily loaded at $11-14$ requests per second, with an average response time of 78 msec. Afterwards at S_2, a major part of load has been shifted to newly created instances, and thus the response time is improved.

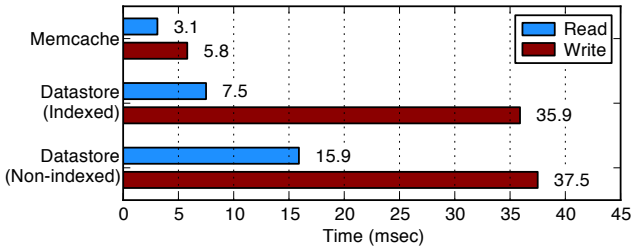

Fig. 15. Performance comparison with different approaches to access a user's spontaneous presence status.

In addition, it is also interesting to investigate the performance gain with detailed tuning for Google App Engine, particularly when combining the distributed *memcache* with the distributed *datastore* to accelerate data access. In Fig. 15, we summarize the time consumed on read/write operations of a user's spontaneous presence status. Surprisingly, we observe almost 6 times of speedup on write operations. As to read operations, we notice that *memcache* is more than twice faster than *datastore* with a proper index in the corresponding data,

and over 5 times faster than *datastore* without the index. This observation also confirms the importance of choosing appropriate keys for *datastore*.

B. Reflex Reflectors across PlanetLab

To evaluate the performance of reflectors in *Reflex*, we deploy the reflector cloud in PlanetLab nodes, which support long-lived distributed service prototypes. Each reflector runs in a PlanetLab "slice," essentially a virtual machine with a resource sandbox on a PlanetLab node.

As discussed in Sec. IV, *Reflex* allows applications to implement custom-tailored reflector selection algorithms, by invoking methods in application-provided *delegate* instances. An application with delay-sensitive sessions may wish to favour reflectors with low latencies, while data-intensive interactive sessions may prefer reflectors with higher throughput and lighter workload. In our experiments, we take minimizing end-to-end delays as a widely accepted scenario, and implement an intuitive solution in which each user *greedily* and *dynamically* selects the reflector with the highest preference, *i.e.*, the one that provides a relay path with the shortest end-to-end delay. Although only one relay path is chosen to be active, nodes keep other idle paths alive, and delay measurements are periodically probed over these idle paths. Once a user finds a "better" reflector that forms a relay path with a shorter delay, it will immediately switch to that reflector.

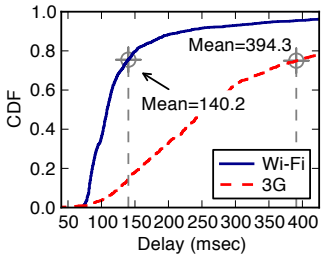

Fig. 16. CDF of end-to-end delay between Wi-Fi/3G users with the greedy algorithm.

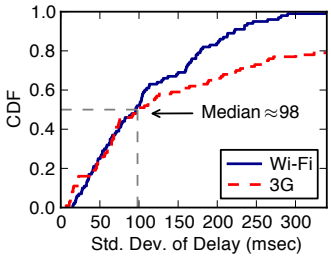

Fig. 17. CDF of standard deviation of end-to-end delays in different Internet access types.

In Fig. 16, our measurement shows that the average end-to-end delay is 140.2 msec and 394.3 msec, respectively, for Wi-Fi and 3G users, which are acceptable in general. With respect to variations of delays over time, we quantify such delay variations in our trace by plotting the CDF of standard deviation of end-to-end delays in different Internet access types in Fig. 17. As shown, in both faster Wi-Fi networks and slower cellular networks, half of the end users experienced less than 98 milliseconds of delay variations.

To measure the performance of reflectors on heavy load scenarios, we perform pressure tests on a typical *Reflex* reflector with a monotonically increasing number of users and forwarding requests, as shown in Fig. 18. We notice that the forwarding rate reaches a ceiling of around 6×10^2 packets/sec, when the CPU is fully utilized. With respect to memory consumption, thanks to our event-driven design, there is no significant increase as more users arrive, and the usage increases only when more unprocessed packets are queued.

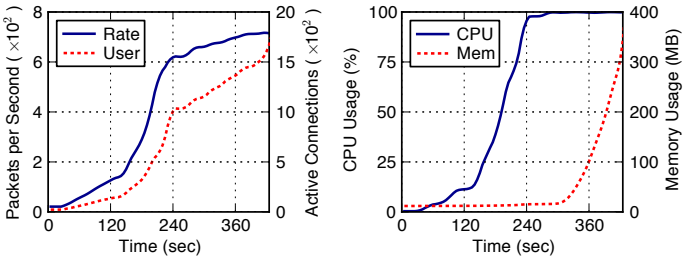

Fig. 18. Pressure testing results on a *Reflex* reflector with an increasing workload.

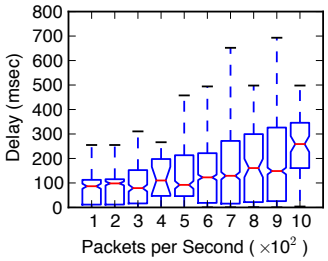

Fig. 19. Impact of reflector workload (in terms of packet forwarding rate) on end-to-end delays.

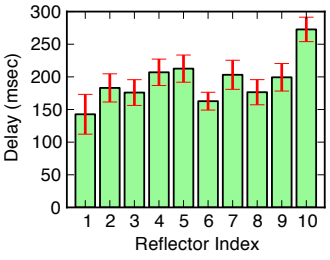

Fig. 20. Performance of 10 typical reflectors (out of 53) deployed in different countries.

In Fig. 19, we further investigate the impact of the reflector workload — in terms of the forwarding rate — on the latency of relay paths over a reflector, *i.e.*, the end-to-end delay between two mobile devices. The box-and-whisker plotting reveals that, when the reflector load is lower than or around 5×10^2 packets/sec, medians (red bars within notched boxes) and upper quartiles (top of boxes) of end-to-end delay are below 100 msec and 200 msec, respectively. Since the *Reflex* framework is designed to serve mobile users all over the world, we examine the performance of globally distributed reflectors in Fig. 20. In the case of *MusicScore*, as our default reflector selection algorithm prefers low latency relay paths, upper quartiles of delay are roughly balanced and the means are mostly below 300 msec.

C. Reflex with Local Interactions

With respect to the support for local interactions in *Reflex*, we conduct experiments with two iPads within the same local wireless network, and within an active Bluetooth transmission range. In Table II, we summarize statistics of improvement on one of two iPads when local interactions are enabled in a 2-party music composition session in *MusicScore*. Clearly, local interactions have significantly improved the performance of *Reflex* at mobile devices, in terms of throughput, delay, as well as battery life.

In addition, we are interested in the case when a *Reflex* user works as a proxy to help a local node. In our experiments, an iPad connected with a 3G cellular network serves as a proxy for another iPad using *Reflex*. Shown in Fig. 21, we discover

TABLE II
PERFORMANCE IMPROVEMENTS WITH LOCAL INTERACTIONS.

Connectivity	Internet		Local	
	3G	Wi-Fi/DSL	Wi-Fi	Bluetooth
Throughput (KB/s)	92*	482	1,081	69
Delay (msec)	298	116	76	41
Battery life (hours)	6.1	6.9	6.9	7.4

Note: The 3G throughput is constrained by the unlink capacity.

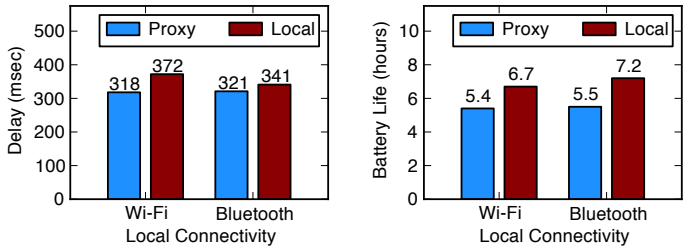

Fig. 21. Performance statistics when a *Reflex* proxy is used.

that the experienced end-to-end delay of the proxy device is only affected by the local device slightly (around 10%), and the battery life is reduced up to 20%. Measurement results also show that the communication overhead at a *Reflex* proxy is around 5.8%.

VI. Related Work

In recent years, there is a strong trend of closely integrating mobile devices with the cloud to provide various feature-rich services. Cuervo *et al.* proposes MAUI to offload computing workload from mobile devices to the cloud to maximize energy saving [8]. CloneCloud designed by Chun *et al.* makes it possible for unmodified mobile applications running in an application-level virtual machine to seamlessly offload part of their execution to the cloud [7]. Wu *et al.* propose a widget-based model for mobile social network service, featuring both high scalability and flexibility [11]. However, to our best of knowledge, no literature has reported system frameworks supporting spontaneous social interactions within the cloud for mobile applications.

In the context of industrial protocols and standards, the Extensible Messaging and Presence Protocol (XMPP) attempts to extend the features of Jabber [9], an open instant messaging and online presence platform, with a pure XML signaling channel that can be used for various applications beyond instant messaging. Jingle [10] was proposed to extend XMPP to support the transmission of large volumes of binary data rather than textual messages, such as voice, video, and file sharing. Signaling and data transmissions in Jingle are carried out in two separate channels. In order to transmit multimedia content, a sender has to first upload its data to a dedicated server, and then send the server address to receivers. Only then can receivers download data from the server. As such, Jingle is not specifically designed for spontaneous social interactive sessions, let alone other features — such as a reusable and flexible API and transparent switching to local interfaces — supported by the *Reflex* framework.

Another related framework may be Apple's *Game Kit*, which supports online presence service using its cloud service called *Game Center*, when a user presents her Apple ID. Game Kit is designed to manage player information in games, such as scoreboards, as well as the exchange of in-game data. When in-game data is to be exchanged, however, all participants of a match have to rely on a full peer-to-peer mesh established among them [3], with no centralized cloud-based infrastructure for data communication. Such a full mesh may suffer from connectivity problems when Network Address Translation (NAT) devices or firewalls block peer-to-peer connections. Without cloud support, it would be difficult to adopt it as a reliable data exchange facility for media-rich content.

VII. Concluding Remarks

What we see today, with online social networks connecting people via status messaging, is just a beginning. It is our hope that a spontaneous social experience in mobile applications can deliver its promise to motivate people to act and create together, solve meaningful problems together, and improve the quality of social wellbeing. In this paper, we present and evaluate the design of *Reflex*, a system framework in the iOS platform, supporting spontaneous presence without requiring explicit user account registration, as well as live interaction sessions using cloud services. Our design in *Reflex* is governed by the principles of *reusability* and *flexibility*. In closing, we are in the hope that this paper only represents the first step towards a mature framework that facilitates spontaneous social interaction within mobile applications, so that users may socially interact with one another to create or consume media content in a spontaneous and transparent fashion, wherever they may be around the world.

References

[1] *geohash*. http://geohash.org/.
[2] *Google App Engine*. http://code.google.com/appengine/.
[3] *Game Kit Programming Guide*. Apple Inc., 2010.
[4] G. Ananthanarayanan and I. Stoica. Blue-Fi: Enhancing Wi-Fi Performance using Bluetooth Signals. In *Proc. ACM MobiSys*, 2009.
[5] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, EECS, UC Berkeley, 2009.
[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Trans. on Computer Systems*, volume 26, 2008.
[7] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: Elastic Execution between Mobile Device and Cloud. *EuroSys*, 2011.
[8] E. Cuervo, A. Balasubramanian, D. ki Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. *ACM MobiSys*, 2010.
[9] P. Saint-Andre. *Extensible Messaging and Presence Protocol (XMPP) Core*. RFC3920, 2004.
[10] P. Saint-Andre. Jingle: Jabber Does Multimedia. *IEEE Multimedia*, 14(1):90–94, 2007.
[11] Z. Wu, C. Zhang, Y. Ji, and H. Wang. Towards Cloud and Terminal Collaborative Mobile Social Network Service. *2010 IEEE International Conference on Social Computing*, pages 623–629, 2010.