Customizable and Extensible Deployment for Mobile/Cloud Applications

Irene Zhang Adriana Szekeres Dana Van Aken Isaac Ackerman
Steven D. Gribble* Arvind Krishnamurthy Henry M. Levy
University of Washington

Abstract

Modern applications face new challenges in managing today's highly distributed and heterogeneous environment. For example, they must stitch together code that crosses smartphones, tablets, personal devices, and cloud services, connected by variable wide-area networks, such as WiFi and 4G. This paper describes Sapphire*, a distributed programming platform that simplifies the programming of today's mobile/cloud applications. Sapphire's key design feature is its distributed runtime system, which supports a flexible and extensible* deployment layer *for solving complex distributed systems tasks, such as fault-tolerance, code-offloading, and caching. Rather than writing distributed systems code, programmers choose* deployment managers *that extend Sapphire's kernel to meet their applications' deployment requirements. In this way, each application runs on an underlying platform that is customized for its own distribution needs.*

1 Introduction

In less than a decade, the computing landscape has undergone two revolutionary changes: the development of small, yet remarkably powerful, mobile devices and the move to massive-scale cloud computing. These changes have led to a shift away from traditional desktop applications to modern mobile/cloud applications.

As a consequence, modern applications have become inherently *distributed*, with data and code spread across cloud backends and user devices such as phones and tablets. Application programmers face new challenges that were visible only to designers of large-scale distributed systems in the past. Among them are coordinating shared data across multiple devices and servers, offloading code from devices to the cloud, and integrating heterogeneous components with vastly different software stacks and hardware resources.

To address these challenges, programmers must make numerous distributed *deployment* decisions, such as:

- Where data and computation should be located
- What data should be replicated or cached
- What data consistency level is needed

These decisions depend on application requirements – such as scalability and fault tolerance – which force difficult performance vs. function trade-offs. The dependency between application requirements and deployment decisions leads programmers to mix deployment decisions with complex application logic in the code, which makes mobile/cloud applications difficult to implement, debug, maintain, and evolve. Even worse, the rapid evolution of devices, networks, systems, and applications means that the trade-offs that impact these deployment decisions are constantly in flux. For all of these reasons, programmers need *a flexible* system that allows them to easily *create and modify distributed application deployments without needing to rewrite major parts of their application*.

This paper presents Sapphire, a general-purpose distributed programming platform that greatly simplifies the design and implementation of applications spanning mobile devices and clouds. Sapphire removes much of the complexity of managing a wide-area, multi-platform environment, yet still provides developers with the fine-grained control needed to meet critical application needs. A key concept of Sapphire's design is the *separation of application logic from deployment logic*. That is, deployment code is factored out of application code, allowing the programmer to focus on the application logic. At the same time, the programmer has full control over deployment decisions and the flexibility to customize them.

Sapphire's architecture facilitates this separation with a highly extensible distributed kernel/runtime system. At the bottom layer, Sapphire's *Deployment Kernel* (DK) integrates heterogeneous mobile devices and cloud servers through a set of common low-level mechanisms, including best-efforts RPC communication, failure detection, and location finding. Between the kernel and the application is a *deployment layer* – a collection of pluggable *Deployment Manager* (DM) modules that extend the kernel to support application-specific deployment needs, such as replication and caching. DMs are written in a generic, application-transparent way, using interposition to intercept important application events, such as RPC calls. The DK provides a simple yet powerful distributed execution environment and API for DMs that makes them extremely easy to write and extend. Conceptually, Sapphire's DK/DM architecture creates a seamless distributed runtime system that is customized specifically for each application's requirements.

We implemented a Sapphire prototype on Linux servers and Android mobile phones and tablets. The prototype includes a library of 26 Deployment Managers

*Currently at Google.

supporting a wide range of distributed management tasks, such as consistent client-side caching, durable transactions, Paxos replication, and dynamic code offloading between mobile devices and the cloud. We also built 10 Sapphire applications, including a fully featured Twitter clone, a multi-player game, and a shared text editor.

Our experience and evaluation show that Sapphire's extensible three-layer architecture greatly simplifies the construction of both mobile/cloud applications and distributed deployment functions. For example, a single-line application code change – switching from one DM to another – is sufficient to transform a cloud-based multi-player game into a P2P (device-to-device) version that significantly improves the game's performance. The division of function between the DK and DM layers makes deployments extremely easy to code; e.g., the DM to support Paxos state machine replication is only 129 lines of code, an order of magnitude smaller than a C++ implementation built atop an RPC library. We also demonstrate that Sapphire's structure provides fine-grained control over performance trade-offs, delivering performance commensurate with today's popular communication mechanisms like REST.

The next section provides background on current mobile/cloud applications and discusses related work. Section 3 overviews Sapphire and its core distributed runtime system. Section 4 presents the application programming model. Section 5 details the design of the Deployment Kernel, while Section 6 focuses on Deployment Managers, which extend the DK with custom distributed deployment mechanisms. Sapphire's prototype implementation is described in Section 7 and evaluated in Section 8, and we conclude in Section 10.

2 Motivation and Background

Figure 1 shows the deployment of a typical mobile/cloud application. Currently, programmers must deploy applications across a patchwork of user devices, cloud servers, and backend services, while satisfying demanding requirements such as responsiveness and availability. For example, programmers may need to apply caching techniques, perform application-specific code splitting across clients and servers, and develop solutions for fast and convenient data sharing, scalability, and fault tolerance.

Programmers use tools and systems when they match the needs of their application. In some cases an existing system might support an application entirely; for example, a simple application that only requires data synchronization could use a backend storage service like Dropbox [23], Parse [53] or S3 [58]. More complex applications, though, must integrate multiple tools and systems into a custom platform that meets their needs. These systems include server-side storage like Redis [56] or MySQL [49] for fault-tolerance, protocols such as

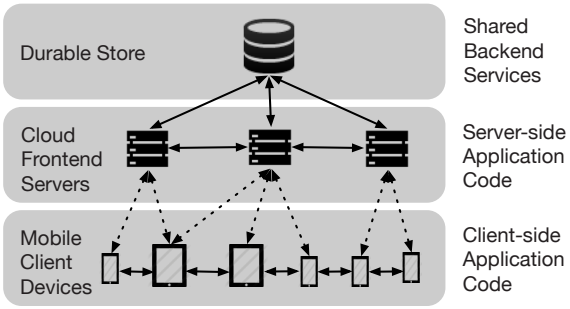

Figure 1: Code for today's applications spans cloud servers and mobile devices. Client-side code runs on varied mobile platforms, while server-side code runs in the cloud, typically using shared backend services like distributed storage.

REST [25] and SOAP [62] or libraries like Java RMI and Thrift [3] for distributed communication, load-balanced servers for scalability, client-side caching for lower wide-area latency, and systems for notification [1], coordination [9, 33], and monitoring [18].

Sapphire provides a flexible environment whose extension mechanism can subsume the functions of many of these systems, or can integrate them into the platform in a transparent way. Programmers can easily customize the runtime system to meet the needs of their applications. In addition, programmers can quickly switch deployment solutions to respond to environment or requirement changes, or simply to test and compare alternatives during development. Finally, Sapphire's Deployment Manager framework simplifies the development or extension of distributed deployment code.

3 Sapphire Overview

Sapphire is a distributed programming platform designed for flexibility and extensibility. In this section, we cover our goals in designing Sapphire, the deployment model that we assume, and Sapphire's system architecture.

3.1 Design Goals

We designed Sapphire with three primary goals:

1. *Create a distributed programming platform spanning devices and the cloud.* A common platform integrates the heterogeneous distributed environment and simplifies communications, code/data mobility, and replication.
2. *Separate application logic from deployment logic.* The application code is focused on servicing client requests rather than distribution. This simplifies programming, evolution, and optimization.
3. *Facilitate system extension and customization.* The delegation of distribution management to an extensible deployment layer gives programmers the flexibility to easily make or change deployment options.

Sapphire is designed to deploy applications across mobile devices and cloud servers. This environment causes

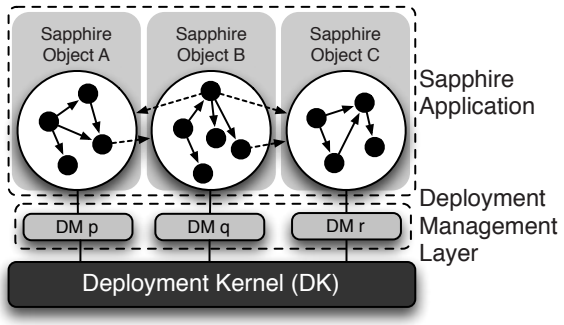

Figure 2: Sapphire runtime architecture. A Sapphire application consists of a distributed collection of Sapphire Objects executing on a distributed Deployment Kernel (DK). A DK instance runs on every device or cloud node. The Deployment Management (DM) layer handles distribution management/deployment tasks, such as replication, scalability, and performance.

significant complexity, as the programmer must stitch together a distributed collection of highly heterogeneous software and hardware components with a broad spectrum of capabilities, while still meeting application goals.

Sapphire is *not* designed for deploying backend services like Spanner [16] or ZooKeeper [33]; its applications interact with such backend services using direct calls, similar to current apps. A Sapphire Deployment Manager can easily integrate a backend service transparently to the application, e.g., using ZooKeeper for coordination or Spanner for fault-tolerance. Sapphire is also not designed for building user interfaces; we expect applications to customize their user interfaces for the devices they employ.

3.2 System Architecture

Figure 2 shows an application-level view of Sapphire's architecture. A Sapphire application, which encompasses all of the client-side and server-side application logic, consists of a collection of *Sapphire Objects* (SOs). Each Sapphire Object functions as a single unit of distribution, like a virtual node. Sapphire Objects in an application share a logical address space that spans all cloud servers and client-side devices. That is, a Sapphire application is written so that all SOs can invoke each other directly through simple location-independent procedure calls.

The bottom layer of Figure 2 is the *Deployment Kernel* (DK), which is a flexible and extensible distributed runtime system. It provides only the most basic distribution functions, including SO addressing and location tracking, best effort RPC-based communication, SO migration, and basic resource management. It does *not* support more complex tasks, such as fault tolerance, failure management, reliability, and consistency. In this way, the DK resembles IP-level network messaging – it is a basic service that relies on higher levels of software to meet more demanding program goals. The kernel is thus deployment agnostic and does not favor (or limit the application to) any specific approaches to deployment issues.

More complex management tasks are supported in the deployment layer by extensions to the DK, called *Deployment Managers* (DMs). Each Sapphire Object can optionally have an attached DM – shown in the middle of Figure 2 – which provides runtime distribution support in addition to the minimal features of the DK. The programmer selects a DM to manage each SO; e.g., he may choose a DM that handles failures to improve fault-tolerance, or one to cache data locally on a mobile device for performance. We have built a library of DMs supporting common distribution tasks used by applications today.

The separation between the DK and DMs provides significant flexibility and extensibility within the Sapphire distributed programming platform. As extensions to the DK, Deployment Managers provide additional distribution management features or guarantees for individual SOs. Often, these features involve performance trade-offs; thus, not every application or every SO will want or need a DM. Finally, by separating application logic (in the application program) from deployment logic (provided by DMs), we greatly reduce application complexity and allow programmers to easily change application deployment or performance behaviors.

4 Programming Model

The Sapphire application programming model is object based and could be integrated with any object-oriented language. Our implementation (Section 7) uses Java.

Sapphire Objects are the key programming abstraction for managing application code and data *locality*. To develop a Sapphire application, the programmer first builds the application logic as a single object-oriented program. He then breaks the application into distributed components by declaring a set of application objects to be Sapphire Objects. Sapphire Objects can still call each other via normal method invocation, however, these calls may now be remote invocations. Finally, the programmer applies Deployment Managers (DMs) to SOs as desired for additional distributed management features. In this section, we will show that the Sapphire programming model provides: (1) ease of programming in a distributed environment, (2) flexibility in deployment, and (3) programmer control over performance.

Defining Sapphire Objects. Programmers define Sapphire Objects as classes using a `sapphireclass` declaration, instead of the standard `class` declaration. As an example, Figure 3 shows a code snippet from our Twitter-clone, BlueBird. All instances of the User class defined here are independent SOs. In this case, the programmer has also specified a DM for the class, called ConsistentCaching, to enhance the object's performance.

SOs can encapsulate internal language-defined objects

```
1  public sapphireclass User uses ConsistentCaching {
2
3      // user handle
4      String username;
5      // people who follow me
6      User[] followers;
7      // people who I follow
8      User[] friends;
9        ....
10     public String getUsername() {
11         return username;
12     }
13     public User[] getMyFollowers() {
14         return followers;
15     }
16     public User[] getPeopleIFollow() {
17         return friends;
18     }
19     public Tweet[] getMyTweets() {
20         return myTweets.getTweets();
21     }
22 }
```

Figure 3: Example Sapphire object from BlueBird.

(Java objects in our system), such as the User string and arrays. These are shown as small solid circles in Figure 2; the solid arrows in the figure are references between internal objects within an SO. SO-internal objects cannot move independently or be accessed directly from outside the SO. The SO is therefore the granularity of distribution and decomposition in Sapphire. Moving an SO always moves all of its internal objects along with it; therefore, the programmer knows that all SO-internal objects will always be co-located with the SO.

A Sapphire Object encapsulates data and computation into a "virtual node" that: (1) ensures that each data/computation unit (a Sapphire Object) will always have its code and data on the same node, (2) lets the system transparently relocate or offload that unit, (3) supports easy replication of units, and (4) provides an easy-to-understand unit of failure and recovery. These benefits make Sapphire Objects a powerful abstraction; using fine-grained programmer-defined Sapphire Objects, instead of a coarse-grained client/server architecture, increases *both* flexibility in distributed deployment and programmer control over performance.

Calling Sapphire Objects. Sapphire Objects communicate using method invocation. The dashed lines in Figure 2 show cross-SO references, which are used to invoke the target SO's public methods. Invocation is location-independent and *symmetric*; it can occur transparently from mobile device to server, from server to device, from device to device, or between servers in the cloud. An SO can be moved by its DM or by the DK as a result of resource constraints on the executing node. Therefore, between two consecutive invocations from SO A to SO B, either or both objects can change location; the DK hides this change from the communicating parties. Invocations can fail, e.g., due to network or node failure; DMs help to handle failure on behalf of SOs.

SOs are passed by reference. All other arguments and return values from SO invocations are passed by value. For example, the return value of `getUsername()` in Figure 3 is a copy of the username object stored inside the SO, while `getMyFollowers()` returns a copy of the array containing references to User SOs. This preserves the encapsulation and isolation properties of Sapphire Objects, since it is impossible to export the address of internal objects within them.

Our goal was to create a uniform programming model integrating mobile devices and the cloud *without* hiding performance costs and trade-offs from the programmer. Therefore, the programmer makes explicit choices in the decomposition of the application into SOs; once that choice is made, the system provides location-independent communication, which simplifies programming in the distributed environment.

Choosing Deployment Managers. Programmers employ the `uses` keyword to specify a DM when defining a Sapphire Object. For example, in Figure 3, the `sapphireclass` declaration (line 1) binds the ConsistentCaching DM to the User class. In this case, every instance of User created by the program will have the ConsistentCaching DM attached to it. It is easy to change the DM binding with a simple change to the sapphireclass definition.

Supporting DMs on a class basis lets programmers specify different features or properties for different application components. While the binding between an SO and its DM could be specified outside of the language (e.g., through a configuration file), we felt that this choice should be visible in the code because deployment decisions about the SO are closely tied to the requirements of an SO.

Sapphire provides a library of standard DMs, and most programmers will be able to choose the behavior they want from the standard library. Additionally, DMs are extensible; we discuss the API for building them in the next section. As programmers can build their own DMs and DMs are designed to be reusable, we expect the library to grow naturally over time.

An SO can have at most one DM, and each instance of the SO must use the same DM. We chose these restrictions for simplicity and predictability, both in the design of applications and DMs. In particular, the behavior of multiple DMs attached to an SO depends on the order in which the functions of the multiple DMs are invoked, and DMs could potentially interfere with each other. For this reason, programmers achieve the same result by explicitly composing DMs using inheritance. This allows the programmer to precisely control the actions of the composed DM. Since instances of the same SO should have the same deployment requirements, we chose not to allow different DMs for different instances of the same SO.

DMs separate management code into generic, reusable

modules that: (1) automatically deploy the application in complex ways, (2) give programmers per-application-component control over deployment trade-offs, and (3) allow programmers to easily change deployment decisions. These advantages make DMs a powerful mechanism for deploying distributed applications.

5 Deployment Kernel

Sapphire's *Deployment Kernel* is a distributed runtime system for Sapphire applications. At a high level, the goal of the DK is to create an integrated execution platform across mobile devices and servers. The key functions provided by the DK include: (1) management and location tracking of Sapphire Objects, (2) location-transparent inter-object communications (RPC), (3) low-level replica support, and (4) services to simplify the writing and execution of Deployment Managers.

A DK instance provides best-effort deployment of a single Sapphire application. It consists of a set of servers that run on every mobile and back-end computing device used by the application, and a centralized Object Tracking System (OTS) for tracking Sapphire Objects.

The Sapphire OTS is a distributed, fault-tolerant coordination service, similar to Chubby [9], ZooKeeper [33] and Tango [4]. The OTS is responsible for tracking Sapphire Objects across DK servers. DK servers only communicate occasionally with the OTS when creating or moving SOs. DK servers do not have to contact the OTS on every RPC because SO references contain a cached copy of the SO's last location,

Each DK server hosts a number of SOs by acting as an event server for the SOs, receiving and dispatching RPCs. The DK server also hosts and manages the DMs for those SOs. DK servers instantiate SOs locally by initializing the SO's memory, creating its DM (which potentially has components on multiple nodes), and registering the SO with the OTS. Once created, the server can move the SO at any time because SO location and movement are invisible to the application.

The DK provides primitive SO scheduling and placement. If a DK server becomes overloaded, it will contact the OTS to find a new server to host the SO, move the SO to the new server, and update the OTS with the SO's new location. The DK API, described in Section 6, provides primitives that allow DMs to express more complex placement and scheduling policies, such as geo-replicated fault-tolerance, load balancing, etc.

To route an RPC to an SO, the calling DK server sends the RPC request to the destination server cached in the SO reference. If the destination no longer hosts the SO, the caller contacts the OTS to obtain the new address. If the destination server is unavailable, the calling server returns an error, because RPC in the DK is always best effort; DMs implement more advanced RPC handling, like retrying RPCs, routing RPCs between replicas, etc.

DK servers are not fault-tolerant: when they fail, they simply reboot. That is, on recovery, DK servers do not recover the SOs that they hosted on failure; they simply register with the OTS and begin hosting new SOs. Failures are entirely handled by DMs. We assume there is a failure detection system, such as FALCON [39], to notify the OTS when servers fail, which will then notify the DMs of the SOs that were hosted on the failed server.

We expect devices to be Internet connected most of the time, since applications today frequently depend on online access to cloud servers. When a device becomes disconnected, its DK server continues to run, but the application will be unable to make or receive remote RPCs. Any SOs hosted on a disconnected device will thus be inaccessible to outside devices and servers. The OTS keeps a list of mobile device IP addresses to quickly re-register SOs hosted on those devices when they reconnect. DMs can provide more advanced offline access.

6 Deployment Managers

A key feature of the Sapphire kernel is its support for the programming and execution of Deployment Managers, which customize and control the behavior of individual SOs in the distributed mobile/cloud environment. The DK provides direct API support for DMs. That API is available to DM developers, who we expect to be more technically sophisticated than application developers, although the DM framework can be used by anyone to customize or build new DMs. As this section will show, DMs can accomplish complex distributed deployment tasks with surprisingly little code. This is due to the careful factoring of function between the DMs and the DK: the DK does the heavy lifting, while the DMs simply tell the DK what to lift through the DK's API.

6.1 DM Library

Sapphire provides programmers with a library of DMs that encompass many management features, including controls over placement and RPC semantics, fault-tolerance, load balancing and scaling, code-offloading, and peer-to-peer deployment. Table 1 lists the DMs that we have built along with a description and the LoC count (from SLOCCount [71]) for each one. We built these DMs both to provide programmers with useful DMs for their applications and to illustrate the flexibility and programming ease of the DM programming framework.

6.2 DM Structure and API

We designed the DM API to provide as minimal an interface as possible while still supporting a wide range of extensions. A DM extends the functionality of the DK to meet the deployment requirements of a specific SO by interposing on DK events for the SO. For example, on an RPC to the SO, the DK will make an upcall into the DM

Table 1: Library of Deployment Managers

Category	Extension	Description	LoC
Primitives	Immutable	Efficient distribution and access for immutable SOs	19
	AtLeastOnceRPC	Automatically retry RPCs for bounded amount of time	27
	KeepInPlace	Keep SO where it was created (e.g., to access device-specific APIs)	15
	KeepInCloud	Keep SO on cloud server (e.g., for availability)	15
	KeepOnDevice	Keep SO on accessing client device and dynamically move	45
Caching	ExplicitCaching	Caching w/ explicit push and pull calls from the application	41
	LeaseCaching	Caching w/ server granting leases, local reads and writes for lease-holder	133
	WriteThroughCaching	Caching w/ writes serialized on the server and stale, local reads	43
	ConsistentCaching	Caching w/ updates sent to every replica for strict consistency	98
Serializability	SerializableRPC	Serialize all RPCs to SO with server-side locking	10
	LockingTransactions	Multi-RPC transactions w/ locking, no concurrent transactions	81
	OptimisticTransactions	Transactions with optimistic concurrency control, abort on conflict	92
Checkpointing	ExplicitCheckpoint	App-controlled checkpointing to disk, revert last checkpoint on failure	51
	PeriodicCheckpoint	Checkpoint to disk every N RPCs, revert to last checkpoint on failure	65
	DurableSerializableRPC	Durable serializable RPCs, revert to last successful RPC on failure	29
	DurableTransactions	Durably committed transactions, revert to last commit on failure	112
Replication	ConsensusRSM-Cluster	Single cluster replicated SO w/ atomic RPCs across at least $f+1$ replicas	129
	ConsensusRSM-Geo	Geo-replicated SO w/ atomic RPCs across at least $f+1$ replicas	132
	ConsensusRSM-P2P	SO replicated across client devices w/ atomic RPCs over $f+1$ replicas	138
Mobility	ExplicitMigration	Dynamic placement of SO with explicit move call from application	20
	DynamicMigration	Adaptive, dynamic placement to minimize latency based on accesses	57
	ExplicitCodeOffloading	Dynamic code offloading with offload call from application	49
	CodeOffloading	Adaptive, dynamic code offloading based on measured latencies	95
Scalability	LoadBalancedFrontEnd	Simple load balancing w/ static number of replicas and no consistency	53
	ScaleUpFrontEnd	Load-balancing w/ dynamic allocation of replicas and no consistency	88
	LoadBalancedMasterSlave	Dynamic allocation of load-balanced M-S replicas w/ eventual consistency	177

for that SO. DMs are implemented as objects, therefore each DM can execute code on each upcall and store state between upcalls.

A DM consists of three component types: the *Proxy*, the *Instance Manager*, and the *Coordinator*. A programmer builds a DM by defining three object classes, one for each type. Since DMs are intended to manage distribution, the DK creates a distributed execution environment in which they operate; i.e., a DM is *itself* distributed and its components can operate on different nodes. When the DK instantiates a Sapphire Object with an attached DM, it also instantiates and distributes the DM's components. The DK provides transparent RPC between the DM components of an SO instance for coordination between components.

Figure 4 shows an example deployment of the DM components for a *single* Sapphire Object A. The DK may instantiate many Proxies and Instance Managers but at most one Coordinator, as shown in this figure. The center box (marked "Instance A") indicates that A has two replicas, marked replica 1 and replica 2. Each replica has its own copy of the Instance Manager. Were the DM to request a third replica of A, the DK would also create a new Instance Manager for that replica. A replica and its Instance Manager are always located on the same node.

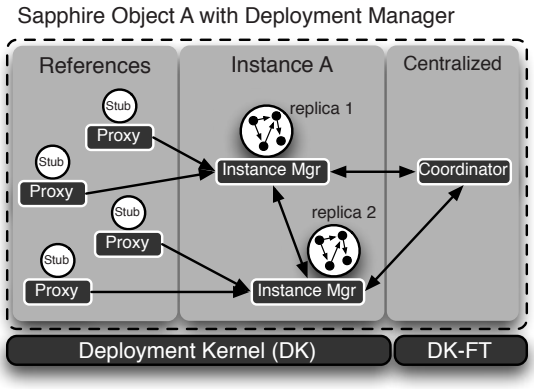

Figure 4: Deployment Manager (DM) organization. The components named *Proxy*, *Instance Mgr*, and *Coordinator* are all part of the DM for one Sapphire Object instance (shown here with two replicas). DK-FT is a set of fault-tolerant DK nodes, which also host the OTS, that support reliable centralized tasks for DMs and the DK.

Each component of the DM is responsible for a particular set of distributed tasks. Proxies are responsible for *caller-side tasks*, like routing method calls. Instance Managers are responsible for *callee-side tasks*, like keeping replicas of the SO synchronized. Note that, due

Table 2: Deployment Managers Upcall API.

Event	Description
`onCreate`	Creation of SO instance
`onRPC`	Method invocation on SO
`onFailure`	Replica failed
`onDestroy`	Coordinator eliminated SO
`onHighLatency`	Avg. RPC latency > limit
`onLowMemory`	Node running out of memory
`onMemberChange`	New replica added to group
`onRefRequest`	Request for an SO reference

to the symmetric nature of SOs, the caller of the method may be on a cloud server and the SO itself may be on a client device. Lastly, the Coordinator is responsible for *centralized tasks* such as failure handling. All three components are optional; a DM can define one or more of the components, and the DK will instantiate only those components that are defined.

The DK completely manages DM components; they run only when invoked, they reside only where the DK places them and are limited to communicating with other components in the same DM instance, which are attached to a single SO. The DK invokes DM components using *upcalls*, which are shown in Table 2. Each component receives a different set of upcalls according to the component's responsibilities. By *interposing* on Sapphire Object events such as method invocations, DMs can implement a variety of distributed management features transparently and generically.

In each upcall, the DM component can perform various management tasks on the SO using a set of primitives supported by the DK. Table 3 lists these primitives. The DM components of an SO instance can communicate directly with each other through a transparent RPC mechanism provided by the DK. Note that the DK supports only the most basic replication functions, namely, creating a new replica for an SO and reporting on replica locations. All decisions about the number of replicas, when to create or delete them, how to synchronize them, and how to handle failures occur at the DM level.

The left-most box in Figure 4 shows four other SOs. Each contains a reference to A, shown as an RPC stub in the figure, to which the DK has attached an instance of A's DM Proxy component. Making an RPC to A through the DK and its DM proceeds as follows. The DK reflects the call via an `onRPC()` upcall to the attached Proxy. The upcall to the Proxy lets A's DM intercept an RPC *on the caller's node* where, for example, it can implement client-local caching. If the Proxy wants to forward the call to replica 1 of A, it simply invokes replica 1's Instance Manager which runs in the same DK server as replica 1. The Instance Manager will pass the RPC through to replica 1 of A.

Because the Proxies and Instance Managers for A are all part of the same Deployment Manager, they all understand whether or not the SO (A, in this case) is replicated, and, if so, *how* that replication is implemented. The choice of which replica to call is made *inside* the DM components, which are aware of each other and can communicate with each other directly through RPCs.

Table 3: DK API for Deployment Managers

Operation	Description
`invoke(RPC)`	Invoke RPC on the local SO
`invoke(SO,RPC)`	Invoke RPC on a specific SO
`getNode()`	Get ID for local node
`getNodes()`	Get list of all nodes
`pin(node)`	Move SO to a node.
`setHighLatency(ms)`	Set limit for RPC latency
`durable_put(SO)`	Save copy of the SO
`durable_get(key)`	Retrieve SO
`replicate()`	Create a replica
`destroyReplica(IM)`	Eliminate a replica
`getReplicas()`	Get list of replicas for SO
`getReplica()`	Get ref to SO instance
`setReplica(SO)`	Set ref to SO instance
`copy(SO)`	Create a copy of the SO instance
`diff(SO,SO)`	Diff two SO instances
`sync(SO)`	Synchronize two SO instances
`getIM()`	Get ref to DM Instance Mgr
`setIM(IM)`	Set reference to DM Instance Mgr
`getCoordinator()`	Get ref to DM Coordinator
`getReference(IM)`	Create DM Proxy for IM
`registerMethod(m)`	Register a custom method for DM
`getRegion()`	Get ID for local region
`getNode()`	Get ID for local node
`pin(region)`	Move SO to region
`pin(node)`	Move SO to node
`getRegions()`	Get list of server regions
`getNodes()`	Get list of nodes in local region

Finally, the DK instantiates one Coordinator for each DM instance, shown in the right-most box of Figure 4. The OTS manages Coordinators, keeping them fault-tolerant and centrally accessible. It is well known that a centralized coordinator can simplify many distributed algorithms (e.g., eliminating the need for leader election). Since the DK needs the OTS to tracking Sapphire Objects, it was easy to provide fault-tolerance for some DMs as well. We do not expect every DM to have a Coordinator, and even if there is a Coordinator, it is used sparingly for management tasks that are easiest handled centrally, such as instantiating new replicas in the event of failures. In this sense, Coordinators are similar to other centralized management systems, like Chubby [9] or ZooKeeper [33].

Programmers can easily extend or compose existing DMs using inheritance. The new DM inherits all of the behavior of the super-DM's Component object classes. The programmer can then override or combine upcalls in each component. While we considered automatic composition,

```
1  public class LeasedCaching extends DManager {
2    public class LCProxy extends Proxy {
3      Lease lease;
4      SapphireObject so;
5
6      public Object onRPC(SapphireRPC rpc) {
7        if (!lease.isValid() || lease.isExpired()) {
8          lease = Sapphire.getReplica().getLease();
9          if (!lease.isValid()) {
10           throw new SONotAvailableException(
11             ''Could not get lease.'');
12         } else {
13           so = lease.getSO();
14         }
15       }
16
17       SapphireObject oldSO = Sapphire.copy(so);
18       Sapphire.invoke(so, rpc);
19       SOStream diff = Sapphire.diff(oldSO, so);
20       if (diff) Sapphire.getReplica().update(diff);
21     }
22   }
23
24   public class LCReplica extends InstanceManager {
25     public synchronized Lease getLease();
26     public synchronized void update(SOStream);
27     // Code for Instance Manager methods
28   }
29 }
```

Figure 5: Example Deployment Manager with arguments.

we believe that the DM programmer should be involved to ensure that the composed DM implements exactly the behavior that the programmer expects. Our experience with composing DMs has shown that the use of inheritance for DM composition is straightforward and intuitive.

6.3 DM Code Example

Figure 5 shows a simplified definition of the LeasedCaching DM that we provide in the Sapphire Library. We include code for the Proxy component and the function declarations from the Instance Manager. This DM does not have a Coordinator because it does not need centralized management.

The LeasedCaching DM is not replicated, so DK will only create one Instance Manager. The Instance Manager hands out mutually exclusive leases to Proxies (which reside with the remote reference to the SO) and uses timeouts to deal with failed Proxies. The Proxy with a valid lease can read or write to a local copy. Read-only operations do not incur communications, which saves latency over a slow network, but updates are synchronously propogated to the Instance Manager in case of Proxy failure.

When the application invokes a method on an SO with this DM attached, the caller's Proxy: (1) verifies that it holds a lease, (2) performs the method call on its local copy, (3) checks whether the object has been modified (using `diff()`), and (4) synchronizes the remote object with its cached copy if the object changed, using an `update()` call to the Instance Manager.

Each Proxy stores the lease in the Lease object (line 3) and a local copy of the Sapphire Object (line 4). If the Proxy does not hold a valid lease, it must get one from the Instance Manager (line 8) before invoking its local SO copy. If the Proxy is not able to get the lease, the DM throws a SONotAvailableException (line 10). The application is prepared for any RPC to an SO to fail, so it will catch the exception and deal with it. The application also knows that the SO uses the LeasedCaching SOM, so it understands the error string (line 11).

If the Proxy is able to get a lease from the Instance Manager, the lease will contain an up-to-date copy of the SO (line 13). The Proxy will make a clean copy of the SO (line 17), invoke the method on its local copy (line 18) and then diff the local copy with the clean copy to check for updates (line 19). If the SO changed, the Proxy will update the Instance Manager's copy of the SO (line 20). The copy and diff is necessary because the Proxy does not know which SO methods might write to the SO, thus requiring an update to the Instance Manager. If the DM had more insight into the SO (i.e., the SO lets the DM know which methods are read-only), we could skip this step.

The example illustrates a few interesting properties of DMs. First, DM code is application agnostic and can perform only a limited set of operations on the SO that it manages. In particular, it can interpose only on method calls to its SO, and it manipulates the managed SO as a black box. For example, there are DMs that automatically cache an SO, but no DMs that cache a part of an SO. This ensures a clean separation of object management code from application logic and allows the DM to be reused across different applications and objects.

Second, a DM cannot span more than one Sapphire Object: it performs operations only on the object that it manages. We chose not to support cross-SO management because it would require the DM to better understand the application; as well, it might cause conflicts between the DMs of different SOs. As a result, there are DMs that provide multi-RPC transactions on a single SO, but we do not support cross-SO transactions. However, the programmer could combine multiple Sapphire Objects into one SO or implement concurrency support at the application level to achieve the same effect.

6.4 DM Design Examples

This section discusses the design and implementation of several classes of DMs from the Sapphire Library, listed in Table 1. Our goal is to show how the DM API can be used to extend the DK for a wide range of distributed management features.

Code-offloading. The code-offloading DMs are useful for compute-intensive applications. The CodeOffloading DM supports transparent object migration based on the performance trade-off between locating an object on a device or in the cloud, while the ExplicitCodeOffloading DM allows the application to decide when to move computation. The ExplicitCodeOffloading DM gives

the application more control than the automated CodeOffloading DM, but is less transparent because the SO must interact with the DM.

Once the DK creates the Sapphire Object on a mobile device, the automated CodeOffloading DM replicates the object in the cloud. The device-side DM Instance Manager then runs several RPCs locally and asks the cloud-side Instance Manager to do the same, calculating the cost of running on each side. An adaptive algorithm, based on Q-learning [70], gradually chooses the lowest-cost option for each RPC. Periodically, the DM retests the alternatives to dynamically adapt to changing behavior since the cost of offloading depends on the type of computation and the network connection, which can change over time.

Peer-to-peer. We built peer-to-peer DMs to support the direct sharing of SOs across client mobile devices *without* needing to go through the cloud. These DMs dynamically place replicas on nodes that contain references to the SO. We implemented the DM using a centralized Coordinator that attempts to place replicas as close to the callers as possible, without exceeding an application-specified maximum number of replicas. We show the performance impact of this P2P scheme in Section 8.

Replication. The Sapphire Library contains three replication DMs that replicate a Sapphire Object across several servers for fault tolerance. They offer guarantees of serializability and exactly-once semantics, along with fault-tolerance. They require that the SO is deterministic and only makes idempotent calls to other SOs.

The Library's replication DMs model the SO as a replicated state machine (RSM) that executes operations on a *master replica*. These DMs all inherit from a common DM that implements the RSM, then extend the common DM to implement different policies for replica placement (e.g., Geo-replicated, P2P).

The RSM DM uses a Coordinator to instantiate the desired number of replicas, designate a leader, and maintain information regarding membership of the replica group. The Coordinator associates an epoch number with this information, which it updates when membership changes.

For each RPC, Instance Managers forward the request to the Instance Manager of the master replica, which logs the RPC and assigns it an ID. The master then sends the ID and epoch number to the other Instance Managers, which accept it if they do not have another RPC with the same ID. If the master receives a response from at least f other Instance Managers, it executes the RPC and synchronizes the state of the SO on the other replicas. If one of the replicas fails, the DK notifies the Coordinator, which allocates a new replica, designates a leader, starts a new epoch, and informs other replicas of the change.

Scalability. To scale Sapphire Objects that handle a large number of requests, the Sapphire Library includes both stateless and stateful scalability DMs. The LoadBalancedFrontEnd DM provides simple load balancing among a set number of replicas. This DM only supports Sapphire Objects that are stateless (i.e., do not require consistency between replicas); however, the SO is free to access state in other Sapphire Objects or on disk. The ScaleUpFrontEnd DM extends the LoadBalancedFrontEnd DM with automatic scale-up, The DM monitors the latency of requests and creates new replicas when the load on the SO and the latency increases. Finally, the LoadBalancedMasterSlave provides scalability for read-heavy workloads by dynamically allocating a number of read-only replicas that receive updates from the master replica. This DM uses the Coordinator to organize replicas and select the master. We show the utility of our scalability DMs in Section 8.

Discussion. The DM's upcall API and its associated DK API are relatively small (only 8 upcalls and 27 DK calls), yet powerful enough to cover a wide range of sophisticated deployment tasks. Most of our DMs are under a hundred lines of code. There are three reasons for this efficiency of expression. First is the division of labor between the DMs and the DK. The DK supports fundamental mechanisms such as RPC, object creation and mobility, and replica management. Therefore, the DK performs the majority of the work in deployment operations, while the DMs simply tell the DK what work to perform.

Second is the availability of a centralized, fault-tolerant Coordinator in the DM environment. This reduces the complexity of many distributed protocols; e.g., in the ConsensusRSM DMs, the Coordinator simplifies consensus by determining the leader and group membership. Our three replication DMs share this code but make different replica placement decisions, meeting different goals and properties with the same mechanism. Inheritance facilitates the composition of new DMs from existing ones; e.g., the DurableTransactions DM builds upon the OptimisticTransactions DM, adding fault-tolerance with only 20 more lines of code.

Finally, the decomposition of applications into Sapphire Objects greatly simplifies DM implementation. We implemented the code-offloading DM in only 95 LoC because we do not have to determine the unit of code to offload dynamically, and because the application provides a hint that the SO is compute-intensive by choosing the DM. In contrast, current code-offloading systems [19, 30, 14] are much more complex because they lack information on application behavior and because the applications are not easily composed into locality units, such as objects.

7 Implementation

Our DK prototype was built using Java to accommodate Android mobile devices. Altogether, the DK consists of 12,735 lines of Java code, including 10,912 lines of

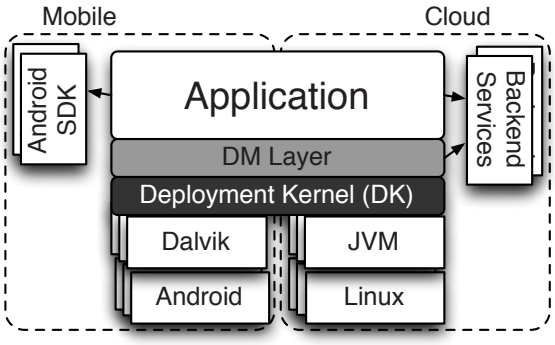

Figure 6: Sapphire application and runtime system.

Apache Harmony RMI code, which we had to port to Dalvik. Dalvik was developed based on Apache Harmony, but does not include an implementation for Java RMI.

Figure 6 shows the prototype's architecture. We used Sun's Java 1.6.0_38 JVM to run Sapphire in the cluster, while the tablets and phones ran Sapphire on the Android 4.2 Dalvik VM. We used Java RMI for low-level RPCs between DK nodes. We used Voldemort [69] as the storage back-end for our checkpointing DMs.

Java RMI provides only point-to-point communication and only supports calls to Java objects that have a special Java RMI-provided interface. Thus, we could only use Java RMI for low-level communication between DK servers and the OTS. To achieve transparent communication between SOs and between DM components, we built a compiler (862 LoC) that creates stubs for SOs and for DM Instance Managers and Coordinators. Having a stub for each SO allows the DK server to route RPCs and invoke DM components on the callee and caller side. DM Instance Managers and Coordinators also require stubs because the DK needs to be able to support transparent RPC from Instance Managers and Proxies. The compiler generates stubs as Java classes that extend the class of the target object, replacing all method contents with forwarding functions into the DK. A stub is therefore a reference that can be used for transparent communication with the remote object through the DK.

We also rely on Apache Harmony's implementation of RMI serialization – with Java reflection to marshall and unmarshall objects – for sending, diffing and copying objects. We did no optimization of Java RMI at all in this prototype. We could have applied well-known techniques [44, 54, 50] to improve RPC performance and expect to do so in the future; however, as we show in our evaluation, our performance is competitive with widely used client-server mechanisms, such as REST. In order to achieve this performance on mobile devices, we had to fix several bugs that caused performance problems in the Apache Harmony RMI code that we ported to Android.

Our prototype does not currently include secure communication between DK servers. Java RMI supports SSL/TLS, so our prototype could easily support encrypted communication between DK servers. We would also require an authentication mechanism for registering DK servers on mobile devices, like Google SSO [28].

In today's applications, mechanisms such as access control checks are typically provided by the application. With a unified programming platform like Sapphire, it becomes possible to move security mechanisms into the platform itself. While this discussion is outside the scope of the paper, we are currently exploring the use of information flow control-based protection for mobile/cloud applications in the context of Sapphire's object and DK/DM structure.

8 Experience and Evaluation

This section presents qualitative and quantitative evaluations of Sapphire. We first describe our experience building new applications and porting applications to Sapphire. Second, we provide low-level DK performance measurements, and an evaluation of several DMs and their performance characteristics. Our experience demonstrates that: (1) Sapphire applications are easy to build, (2) the separation of application code and deployment code, along with the use of symmetric (i.e., non-client-server) communication, maximizes flexibility and choice of deployment for programmers, and (3) Deployment Managers can be used effectively to improve performance and scalability in a dynamic distributed environment.

8.1 Applications

We consider the design and implementation of several Sapphire applications with respect to three objectives:

- **Development Ease:** It should be easy to develop mobile/cloud applications either from scratch or by porting non-distributed mobile device applications to Sapphire. Furthermore, it should be possible to write application code without explicitly addressing distribution management.
- **Deployment Flexibility:** The programmer should be able to choose from alternative distribution management schemes and change deployment decisions without rewriting application code.
- **Management Code Generality:** It should be possible to build generic distribution management components that can be used widely both within an application and across different applications.

Table 4 lists several applications that we built or ported, along with their LoC. We built three applications from scratch: an online to-do list, a collaborative text and table editor, and a multi-player game. We also built a fully-featured Twitter clone, called BlueBird, and paired it with the front-end UI from Twimight [68], an open-source Android Twitter client. The table also lists six non-distributed, compute-intensive applications that

Table 4: Sapphire applications. We divide each application into front-end code (the UI) and back-end code (application logic). The source column indicates whether we developed new native Sapphire code or ported open-source code to Sapphire.

	Back-end		Front-end	
Application	**Source**	**LoC**	**Source**	**LoC**
To Do List	Native	48	Native	132
Text/Table Editor	Native	409	Native	533
Multi-player Game	Native	588	Native	1,186
BlueBird	Native	783	Ported	13,009
Sudoku Solver	Ported	76	-	-
Regression	Ported	348	-	-
Image Recognition	Ported	102	-	-
Physics Engine	Ported	108	-	-
Calculus	Ported	818	-	-
Chess AI	Ported	427	-	-

we ported to Sapphire.

Development Ease. It took relatively little time and programming experience to develop Sapphire applications. In particular, the existence of a DM library lets programmers write application logic without needing to manage distribution explicitly. Two applications – the multi-player game and collaborative editor – were written by undergraduates who had never built mobile device or web applications and had little distributed systems experience. In under a week, each student wrote a working mobile/cloud application of between 1000 and 1500 lines of code consisting of five or six Sapphire Objects spanning the UI and Sapphire back-end.

Porting existing applications to Sapphire was easy as well. For the compute-intensive applications, a single line change was sufficient to turn a Java object into a distributed SO that could adaptively execute either on the cloud or the mobile device. We did not have to handle failures because the CodeOffload DM hides them by transparently re-executing the computation locally when the remote site is not available. An undergraduate ported all six applications – and implemented the CodeOffload DM as well – in less than a week.

Our largest application was BlueBird, a Twitter clone that was organized as ten Sapphire Objects: Tweet, Tag, TagManager, Timeline, UserTimeline, HomeTimeline, MentionsTimeline, FavoritesTimeline, User and UserManager. We implemented all Twitter functions except for messaging and search in under 800 lines. In comparison, BigBird [22], an open-source Twitter clone, is 2563 lines of code, and Retwis-J [38], which relies heavily on Redis search functionality, is 932 lines of code.

Distributed mobile/cloud applications must cope with the challenges of running on resource-constrained mobile devices, unreliable cloud servers, and high-latency, wide-area links. Using Sapphire, these challenges are handled by selecting DMs from the DM library, which greatly simplifies the programmer's task and makes it easy to develop and test alternative deployments.

Deployment Flexibility. Changing an SO's DM, which changes its distribution properties, requires only a one-line code change. We made use of this property throughout the development of our applications as we experimented with our initial distribution decisions and tried to optimize them.

In BlueBird, for example, we initially chose not to make Tweet and Tag into SOs; since these objects are small and immutable, we thought they did not need to be independent, globally shared objects. Later, we realized that it would be useful to refer directly to Tweets and Tags from Timeline objects rather than accessing them through another SO. We therefore changed them to SOs – a trivial change – and then employed ExplicitCaching for both of them to reduce the network delay for reads of the tweet or tag strings.

As another example, we encountered a deployment decision in the development of our multi-player game. The Game object lasts only for the duration of a game and can be accessed only from two devices used to play. Since the object does not need high reliability or availability, it can be deployed in any number of ways: on a server, on one of the devices, or on both devices. We first deployed the Game object on a cloud server and then decided to experiment with peer-to-peer alternatives. Changing from the cloud deployment to peer-to-peer using the KeepOnDevice and ConsensusRSM-P2P managers in our DM library required only a *single line change*, and improved performance (see Section 8.4) and allowed games to continue when the server is unavailable. In contrast, changing an application for one of today's systems from a cloud deployment to a peer-to-peer mobile device deployment would require significant application rewriting (and might even be impossible without an intermediary cloud component due to the client-server nature of existing systems).

Management Code Generality. We applied several DMs to multiple SOs within individual applications and across applications. For example, many of our applications have an object that is shared among a small number of users or devices (e.g., ToDoList, Document, etc.). To make reads faster while ensuring that users see immediate updates, we used the ConsistentCaching DM for all of these applications. Without the DM structure, the programmers would have to write the caching and synchronization code explicitly for each case.

Even within BlueBird, which has 10 Sapphire Object types, we could reuse several DMs. If the deployment code for each BlueBird SO had to be implemented in the application, the application would grow by at least

800 LoC, more than doubling in size! This number is conservative: it assumes the availability of the DM API and the DK for support. Without those mechanisms, even more code would be required.

8.2 Experimental Setup

Our experiments were performed on a homogeneous cluster of server machines and several types of devices (tablets and phones). Each server contained 2 quad-core Intel Xeon E5335 2.00GHz CPUs with 8GB of DRAM running Ubuntu 12.04 with Linux kernel version 3.2.0-26. The devices were Nexus 7 tablets, which run on a 1.3 GHz quad-core Cortex A9 with 1 GB of DRAM, and Nexus S phones with a 1 GHz single-core Hummingbird processor and 512MB of DRAM. The servers were all connected to one top-of-rack switch. The devices were located on the same local area network as the servers, and communicated with the server either through a wireless connection or T-mobile 3G links.

8.3 Microbenchmarks

We measured the DK for latency and throughput using closed-loop RPCs. Latencies were measured at the client. Before taking measurements, we first sent several thousand requests to warm up the JVM to avoid the effects of JIT and buffering optimizations.

RPC Latency Comparison. We compared the performance of Sapphire RPC to Java RMI and to two widely used communication models: Thrift and REST. Apache Thrift [3] is an open-source RPC library used by Facebook, Cloudera and Evernote. REST [25] is a popular low-level communication protocol for the Web; many sites have a public REST API, including Facebook and Twitter. We measured REST using a Java client running the standard `HttpURLConnection` class and a PHP script running on Apache 2.2 for method dispatch.

Table 5 shows request/response latencies for intra-node (local), server-to-server, tablet-to-server, server-to-tablet and tablet-to-tablet communications on null requests for all four systems. While Thrift was slightly faster in all cases, Java RMI and Sapphire were comparable and were both faster than the Java REST library.

Sapphire uses Java RMI for communication between DK servers; however, we dispatch method calls to SOs through the DK. This additional dispatch caused the latency difference between Java RMI and Sapphire RPC. The extra cost was primarily due to instantiating and serializing Sapphire's RPC data object (which is not required for a null Java RMI RPC). We could reduce this cost by using a more efficient RPC and serialization infrastructure, such as Thrift.

Note that even without optimization, Sapphire was faster than REST, which is probably the most widely used communication framework today. Furthermore, we could not show REST performance for server-to-tablet and tablet-to-tablet because REST's client-server architecture cannot accept HTTP requests on the tablet. Thus, REST can be used only for tablet-to-server communication, requiring the application to explicitly manage communication forms such as server-to-client or client-to-client.

Table 5: Request latencies (ms) for local, server-to-server, tablet-to-server, server-to-tablet and tablet-to-tablet. Note that REST does not support communication to tablets.

RPC Protocol	Local	S→S	T→S	S→T	T→T
Sapphire	0.08	0.16	5.9	3.4	12.0
Java RMI	0.05	0.12	4.6	2.0	7.2
Thrift	0.04	0.11	2.0	2.0	3.6
REST	0.49	0.64	7.9	-	-

Throughput Comparison. We measured request throughput for the Sapphire DK and Java RMI. The results (Figure 7) showed similar throughput curves, with Java RMI object throughput approximately 15% higher than that for Sapphire Objects. This is because Sapphire null RPCs are not truly empty: they carry a serialized structure telling the DK how to direct the call. To break the cost down further, we measured the throughput of a Java RMI carrying a payload identical to that of the Sapphire null RPC. This reduced the throughput difference to 3.6%; this 3.6% is the additional cost of Sapphire's RPC dispatching in the DK, with the remainder due to the cost of serialization for the dispatching structure. Again, there are many ways to reduce the cost of this communication in Sapphire, but we leave those optimization to future work.

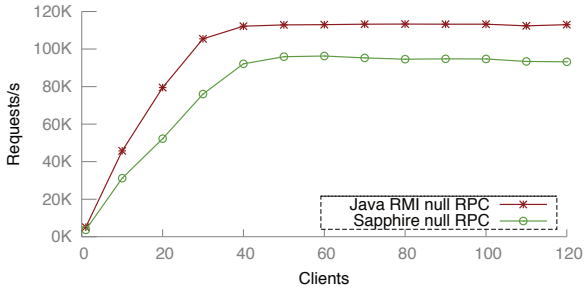

Figure 7: Throughput of a Sapphire Object versus an RMI Object.

Sapphire DK Operation Cost. We measured the latency of several DK services. DK call latency depends on the size and complexity of the object, since we use Java serialization. Table 6 shows latency results for creating, replicating, and moving SOs on servers and tablets. Operation latencies were low when executed on cloud servers. Tablets were considerably slower than cloud servers. However, we expect most management operations such as these to be performed in the cloud (i.e., we do not expect tablets to create large numbers of SOs).

The SO instantiation process can be expensive because

Table 6: Sapphire DK API latencies (ms).

	create		replicate		move (over WiFi)			
Object	**S**	**T**	**S**	**T**	**S→S**	**T→S**	**S→T**	**T→T**
Table	1.1	28	0.5	15	1.9	42	16	66
Game	1.1	29	0.5	16	2.1	49	19	67
TableMgr	1.1	27	0.6	18	2.2	50	16	78

the DK must create several objects locally: the SO, the SO stub, the DM Proxy and the DM Instance Manager. The DK must also create the DM Coordinator remotely on a DK-FT node and register the SO with the OTS. Communication with the DK-FT node and the OTS accounted for nearly half the instantiation latency.

8.4 Deployment Manager Performance

We measured the performance of five categories of DMs: caching, replication, peer-to-peer, mobility, and scalability. Our goal was to examine their effectiveness as extensions to the DK and the costs and trade-offs of employing different DMs.

Caching. We evaluated two caching DMs: LeaseCaching and ConsistentCaching. As expected, caching significantly improved the latency of reads in both cases. For the TodoList SO, which uses the LeaseCaching DM, caching reduced read latency from 6 ms to 0.5 ms, while write latency increased from 6.1 ms to 7.5 ms. For the Game SO, which uses the ConsistentCaching DM, all read latencies decreased, from 7-13 ms to 2-3 ms. With consistent caching, the write cost to keep the caches and cloud synchronized was significant, increasing from 29 ms to 77 ms. Overhead introduced by the DM was due to the use of serialization to determine read vs. write operations. For writes, the whole object was sent to be synchronized with the cloud, instead of a compact patch.

Code offloading. We measured our ported, compute-intensive applications with the CodeOffloading DM for the Nexus 7 tablet and the Galaxy S smartphone. Figure 8 shows the latencies for running each application locally on the device (shown as Base), offloaded to the cloud over WiFi, and offloaded over 3G. The offloading trade-offs varied widely across the two platforms due to differences in CPU speed, wireless, and cellular network card performance. For example, for the Calculus application, cloud offloading was better for the phone over both wireless and 3G; however, for the tablet it was better only over wireless. For the Physics engine, offloading was universally better, but it was particularly significant for the mobile device, which was not able to provide real-time simulation without code offloading.

These cross-platform differences in performance show the importance of flexibility. An automated algorithm cannot always predict when to offload and can be costly. Therefore, it is important for the programmer to be able to easily change deployment to adapt to new technologies.

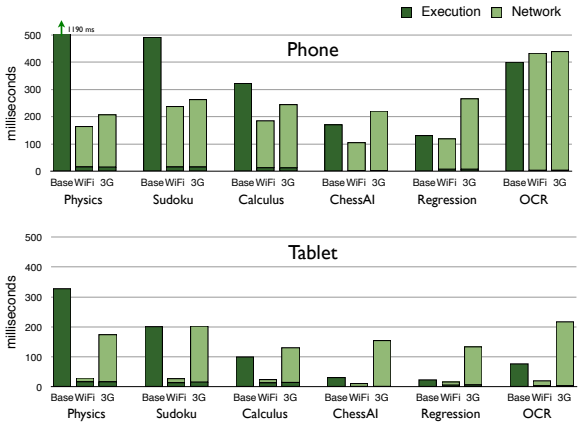

Figure 8: Code offloading performance.

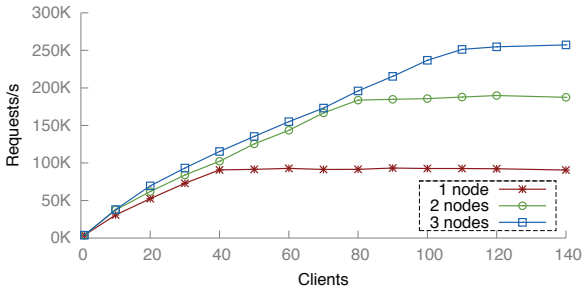

Figure 9: Effects of applying the LoadBalancedFrontEnd DM.

Scalability. We built the LoadBalancedFrontEnd DM to scale a stateless SO under heavy load. The DM creates a given number of non-consistent replicas of an SO and assigns clients to the replicas in a round-robin fashion. Figure 9 shows the throughput of the SO serving null RPCs when the DM creates up to 3 replicas. Throughput scaled linearly with the number of replicas until the network saturated at 257,365 requests/second.

Peer-to-Peer Deployments. Sapphire lets programmers move objects easily between clients and servers, enabling P2P deployments that would be difficult or impossible in existing systems. We measured three deployments for the Game SO from our multi-player game: (1) without a DM, which caused Sapphire to deploy the SO on the server where it is created; (2) with the KeepOnDevice DM, which dynamically moved the Game object to a device that accessed it; and (3) with the ConsensusRSM-P2P DM, which created synchronized replicas of the Game SO on the callers' devices.

For each deployment, Figure 10 shows the latency of the game's read methods (`getScrambleLetters()`, `getPlayerTurn()` and `getLastRoundStats()`) and write methods (`play()` and `pass()`). With the Game SO in the cloud, read and write latencies were high for both players. With the KeepOnDevice DM, the read and write latencies were extremely low for the device hosting the SO, but somewhat higher for the other player, compared to the cloud version. Finally, with the ConsensusRSM-P2P DM, read latencies were much

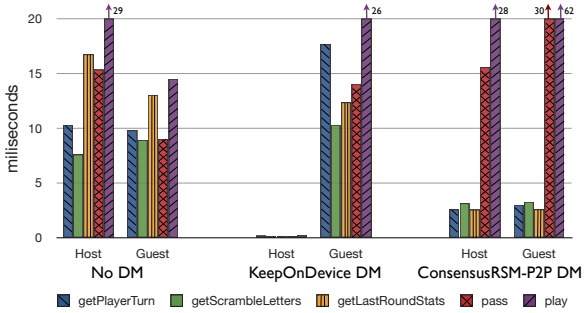

Figure 10: Multi-player Game: different deployment schemes.

lower for both devices, while write latencies were higher. In our scenario, the two tablets and the server were on the same network. In cases where the two players are close on the network and far from the server, the peer-to-peer DMs would provide a valuable deployment option.

With the DMs, no cloud servers were needed to support the Game SO; this reduced server load, but Game SOs were no longer available if the hosting device were disconnected. This experiment shows the impact of different deployment options and the benefit of being able to flexibly choose alternative deployments to trade off application performance, availability, and server load.

9 Related Work

Researchers have built many systems to help applications cope with deployment issues. Code-offloading systems, like COMET [30], MAUI [19], and CloneCloud [14], automatically offload computationally intensive tasks from mobile devices to cloud servers. Distributed storage systems [21, 12, 16] are a popular solution for server-side scalability, durability and fault-tolerance. Systems like PADS [7], PRACTI [6] and WheelFS [65] explored configurable deployment of application *data* but not runtime management of the entire application. Systems like Bayou [67], Cimbiosys [55] and Simba [2] offer client-side caching and offline access for weakly connected environments. Each of these systems only solves a subset of the deployment challenges that mobile/cloud applications face. Sapphire is the first distributed system to provide a unified solution to deployment for mobile/cloud applications.

When building Sapphire's DM library, we drew inspiration from existing mobile/cloud deployment systems, including those providing: wide-area communication [34], load-balancing [31, 72], geographic replication [43, 63], consensus protocols [37, 52], and DHTs [64, 57, 45].

Similar to our goal with Sapphire, previous language and compiler systems have tried to unify the distributed environment. However, unlike Sapphire, these solutions have no flexibility. They either make all deployment decisions for the application – an approach that doesn't work for the wide range of mobile/cloud requirements – or they leave all deployment up to the programmer. Compilers like Coign [32], Links [15], Swift [13] and Hop [60] automatically partition applications, but give programmers no control over performance trade-offs. Single language domains like Node.js [51] and Google Web Toolkit [29] create a uniform programming language across browsers and servers, but leave deployment up to the application. For mobile devices, MobileHTML5 [47], MobiRuby [48] and Corona [17] support a single cross-platform language. Sapphire supports a more complete cross-platform environment, but programmers can select deployments from an extensive (and extensible) library.

The DK's single address space and distributed object model are related to early distributed programming systems such as Argus [41], Amoeba [66] and Emerald [35]. Modern systems like Orleans [10] and Tango [4] provide cloud- or server-side services. Fabric [42] extends the work in this space with language abstractions that provide security guarantees. These systems were intended for homogeneous, local-area networks, so do not have the customizability and extensibility of the Sapphire DK.

Overall, existing or early distributed programming systems are not *general-purpose, flexible* or *extensible* enough to support mobile/cloud application requirements. Therefore, in designing Sapphire, we drew inspiration from work that has explored customizability and extensibility in other contexts: operating systems [24, 8, 26, 59, 40], distributed storage [7, 20, 65, 61, 27], databases [11, 5], and routers and switches [36, 46].

10 Conclusion

This paper presented Sapphire, a system that simplifies the development of mobile/cloud applications. Sapphire's Deployment Kernel creates an integrated environment with location-independent communication across mobile devices and clouds. Its novel deployment layer contains a library of Deployment Managers that handle application-specific distribution issues, such as load-scaling, replication, and caching. Our experience shows that Sapphire: (1) greatly eases the programming of heterogeneous, distributed cloud/mobile applications, (2) provides great flexibility in choosing and changing deployment decisions, and (3) gives programmers fine-grained control over performance, availability, and scalability.

Acknowledgements

This work was supported by the National Science Foundation (grants CNS-0963754, CNS-101647, CSR-1217597), an NSF Graduate Fellowship, the ARCS Foundation, an IBM PhD Scholarship, Google, and the Wissner-Slivka Chair in Computer Science & Engineering. We thank our shepherd Doug Terry and the reviewers for their helpful comments on the paper. Finally, we'd like to thank the UW Systems lab for their support and feedback throughout the project.

References

[1] A. Adya, G. Cooper, D. Myers, and M. Piatek. Thialfi: A client notification service for internet-scale applications. In *Proc. of SOSP*, 2011.

[2] N. Agrawal, A. Aranya, and C. Ungureanu. Mobile data sync in a blink. In *Proc. of HotStorage*, 2013.

[3] Apache. Apache Thrift, 2013. http://thrift.apache.org.

[4] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. In *Proc. of SOSP*, 2013.

[5] D. Batoory, J. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. Twichell, and T. Wise. Genesis: An extensible database management system. *IEEE Transactions on Software Engineering*, 1988.

[6] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. Practi replication. In *Proc. of NSDI*, 2006.

[7] N. M. Belaramani, J. Zheng, A. Nayate, R. Soulé, M. Dahlin, and R. Grimm. PADS: A policy architecture for distributed storage systems. In *Proc. of NSDI*, 2009.

[8] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *Proc. of SOSP*, 1995.

[9] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. of OSDI*, 2006.

[10] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: cloud computing for everyone. In *Proc. of SOCC*, 2011.

[11] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. *Object and file management in the EXODUS extensible database system*. Computer Sciences Department, University of Wisconsin, 1986.

[12] 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. *ACM Transactions on Computer Systems*, 2008.

[13] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proc. of SOSP*, 2007.

[14] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: Elastic execution between mobile device and cloud. In *Proc. of EuroSys*, 2011.

[15] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Proc. of FMCO*, 2006.

[16] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proc. of OSDI*, 2012.

[17] Corona SDK, 2013. http://www.coronalabs.com/.

[18] J. Cowling, D. R. Ports, B. Liskov, R. A. Popa, and A. Gaikwad. Census: Location-aware membership management for large-scale distributed systems. *Proc. of USENIX ATC*, 2009.

[19] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: making smartphones last longer with code offload. In *Proc. of MobiSys*, 2010.

[20] M. Dahlin, L. Gao, A. Nayate, A. Venkataramana, P. Yalagandula, and J. Zheng. PRACTI replication. In *Proc. of NSDI*, 2006.

[21] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. of SOSP*, 2007.

[22] D. Diephouse and P. Brown. Building a highly scalable, open source Twitter clone, 2009. http://fr.slideshare.net/multifariousprb/building-a-highly-scalable-open-source-twitter-clone.

[23] Dropbox, 2013. http://dropbox.com.

[24] D. R. Engler, M. F. Kaashoek, et al. Exokernel: An operating system architecture for application-level resource management. In *Proc. of SOSP*, 1995.

[25] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[26] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proc. of SOSP*, 1997.

[27] R. Geambasu, A. A. Levy, T. Kohno, A. Krishnamurthy, and H. M. Levy. Comet: An active distributed key-value store. In *Proc. of OSDI*, 2010.

[28] 2013. https://developers.google.com/google-apps/marketplace/sso.

[29] Google web toolkit. https://developers.google.com/web-toolkit/, October 2012.

[30] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. COMET: Code offload by migrating execution transparently. In *Proc. of OSDI*, 2012.

[31] HAProxy: A reliable, high-performance TCP/HTTP load balancer, 2013. http://haproxy.1wt.eu/.

[32] G. C. Hunt and M. L. Scott. The coign automatic distributed partitioning system. In *Proc. of OSDI*, 1999.

[33] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proc. of USENIX ATC*, 2010.

[34] A. D. Joseph, A. F. de Lespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: a toolkit for mobile information access. In *Proc. of SOSP*, 1995.

[35] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. In *Proc. of SOSP*, 1987.

[36] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. In *Proc. of SOSP*, 1999.

[37] L. Lamport. Paxos made simple. *ACM Sigact News*, 2001.

[38] C. Leau. Spring Data Redis - Retwis-J, 2013. http://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current/.

[39] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the falcon spy network. In *Proc. of SOSP*, 2011.

[40] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in Hydra. In *Proc. of SOSP*, 1975.

[41] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. In *Proc. of SOSP*, 1987.

[42] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: A platform for secure distributed computation and storage. In *Proc. of SOSP*, 2009.

[43] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proc. of SOSP*, 2011.

[44] J. Maassen, R. Van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for parallel programming. *ACM Transactions on Programming Languages and Systems*, 2001.

[45] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. of IPTPS*, 2002.

[46] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 2008.

[47] Moblie HTML5, 2013. http://mobilehtml5.org.

[48] MobiRuby, 2013. http://mobiruby.org/.

[49] MySQL, 2013. http://www.mysql.com/.

[50] C. Nester, M. Philippsen, and B. Haumacher. A more efficient RMI for Java. In *Proc. of Java Grande*, 1999.

[51] Node.js, 2013. http://nodejs.org/.

[52] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proc. of PODC*, 1988.

[53] Parse, 2013. http://parse.com.

[54] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 2000.

[55] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-sullivan, T. Wobber, C. C. Marshall, and A. Vahdat. Cimbiosys: A platform for content-based partial replication. In *Proc. of NSDI*, 2009.

[56] Redis: Open source data structure server, 2013. http://redis.io/.

[57] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proc. of Middleware*, 2001.

[58] Amazon S3, 2013. http://aws.amazon.com/s3/.

[59] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. of OSDI*, 1996.

[60] M. Serrano, E. Gallesio, and F. Loitsch. Hop: a language for programming the web 2.0. In *OOPSLA Companion*, 2006.

[61] A. Siegel, K. Birman, and K. Marzullo. Deceit: A flexible distributed file system. In *Proc. of the Workshop on the Management of Replicated Data*, 1990.

[62] Simple object access protocol. http://www.w3.org/TR/soap/.

[63] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proc. of SOSP*, 2011.

[64] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of SIGCOMM*, 2001.

[65] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, M. F. Kaashoek, and R. Morris. Flexible, wide-area storage for distributed systems with WheelFS. In *Proc. of NSDI*, 2009.

[66] A. S. Tanenbaum, R. Van Renesse, H. Van Staveren, G. J. Sharp, and S. J. Mullender. Experiences with the Amoeba distributed operating system. *Commun. ACM*, 1990.

[67] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proc. of SOSP*, 1995.

[68] Twimight open-source Twitter client for Android, 2013. http://code.google.com/p/twimight/.

[69] Voldemort: A distributed database, 2013. http://www.project-voldemort.com/voldemort/.

[70] C. Watkins and P. Dayan. Q-learning. *Machine Learning*, 1992.

[71] D. A. Wheeler. SLOCCount, 2013. http://www.dwheeler.com/sloccount/.

[72] Zen load balancer, 2013. http://www.zenloadbalancer.com/.