Slingshot: Deploying Stateful Services in Wireless Hotspots

Ya-Yunn Su and Jason Flinn
Department of Electrical Engineering and Computer Science
University of Michigan

Abstract

Given a sufficiently good network connection, even a handheld computer can run extremely resource-intensive applications by executing the demanding portions on a remote server. At first glance, the increasingly ubiquitous deployment of wireless hotspots seems to offer the connectivity needed for remote execution. However, we show that the backhaul connection from the hotspot to the Internet can be a prohibitive bottleneck for interactive applications. To eliminate this bottleneck, we propose a new architecture, called Slingshot, that replicates remote application state on surrogate computers co-located with wireless access points. The first-class replica of each application executes on a remote server owned by the handheld user; this offers a safe haven for application state in the event of surrogate failure. Slingshot deploys second-class replicas on nearby surrogates to improve application response time. A proxy on the handheld broadcasts each application request to all replicas and returns the first response it receives. We have modified a speech recognizer and a remote desktop to use Slingshot. Our results show that these applications execute 2.6 times faster with Slingshot than with remote execution.

1 Introduction

Creating applications that execute on small, mobile computers is challenging. On one hand, the size and weight constraints of handheld and similar computers limit their processing power, battery capacity, and memory size. On the other hand, user's appetites are driven by the applications that run on desktops; these often require more resources than a handheld provides. A solution to this challenge is remote execution using wireless networks to access compute servers; this combines the mobility of handhelds and the processing power of desktops.

Although Internet connectivity is increasingly ubiquitous due to widespread deployment of wireless hotspots, the backhaul connections between hotspots and the Internet are communication bottlenecks. The uplink bandwidth from a wireless hotspot can be quite limited (e.g. 1.5 Mb/s for a T1 line). Further, this bandwidth must be shared by all hotspot users. The network round-trip time between a hotspot and a remote server may be large due to the use of firewalls and other middleboxes, as well as the vagaries of Internet routing. For interactive applications such as speech recognition and remote desktops, the combination of high latency and low bandwidth is prohibitive; mobile users cannot achieve acceptable response times when communicating with remote servers.

In this paper, we describe Slingshot, a new architecture for deploying mobile services at wireless hotspots. Slingshot replicates applications on *surrogate computers* [1] located at hotspots. A first-class replica of each application executes on a remote server owned by the mobile user. Slingshot instantiates second-class replicas on surrogates at or near the hotspot where the user is located. A proxy running on a handheld broadcasts each application request to all replicas; it returns the first response it receives to the application. Second-class replicas improve interactive response time since they are reachable through low-latency, high-bandwidth connections (e.g. 54 Mb/s for 802.11g). At the same time, the first-class replica is a trusted repository for application state that is not lost in the event of surrogate failure.

Slingshot also simplifies surrogate management. It uses virtual machine encapsulation to eliminate the need to install application-specific code on surrogates. Further, replication prevents the loss of application state when a surrogate crashes or even permanently fails. The performance impact of surrogate failure is mitigated by other replicas, which continue to service client requests.

The harnessing of surrogate computation is a multifaceted problem with many challenges. This paper addresses several of these challenges, including improving interactive response time, hiding the perceived cost of migration, recovering from surrogate failure, and simplifying surrogate management. It also presents concrete results that measure the potential benefit of surrogate computation for stateless and stateful applications. Other challenges remain to be addressed. Slingshot does not yet address privacy concerns, provide protocols for

secure replica management, manage surrogate load, or decide when to instantiate and destroy replicas.

We have implemented two Slingshot services: a speech recognizer and a remote desktop. Our results show that instantiating a second-class replica on a surrogate lets these applications run 2.6 times faster. Our results also show that replication lets Slingshot move services between surrogates with little user-perceived latency and recover gracefully from surrogate failure.

2 Design principles

We begin by discussing the three principles we followed in the design of Slingshot.

2.1 Location, location, location

Server location can be critical to the performance of remote execution. Consider a handheld connected to the Internet at a wireless hotspot. If the handheld executes code on a remote server, its network communication not only passes through the wireless medium; it also traverses the hotspot's backhaul connection and the wide-area Internet link. In a typical hotspot, the backhaul connection is the bottleneck. For instance, the nominal bandwidth of a 802.11g network (54 Mb/s) is more than an order of magnitude greater than that of a T1 connection. If the handheld could instead execute code on a server located at the hotspot, it could avoid the communication delay associated with the bottleneck link. For interactive applications that require sub-second response time, server location can make the difference between acceptable and unacceptable performance.

Network latency is also a concern. A server that is nearby in physical distance can often be quite distant in network topology due to the vagaries of Internet routing. Firewalls, VPNs, and NAT middleboxes add additional latency when connections cross administrative boundaries. For mobile users, a journey of only a few hundred yards can dramatically increase the round-trip time for communication with a remote server. In contrast, a server located at the current hotspot is only a network hop away.

2.2 Replicate rather than migrate

The desire to locate services near mobile users implies that services need to move over time. When a handheld user moves to a new location, a surrogate at the new hotspot will often offer better response time than a surrogate at the previous hotspot.

What is the best method to move functionality? One option is migration: suspend the application on the previous surrogate, transmit its state to the new surrogate, and resume it there. This approach has a substantial drawback: the application is unavailable while it is migrating. Slingshot uses an alternative strategy that instantiates multiple replicas of each service. While a new replica is being instantiated, existing replicas continue to serve the user.

Slingshot replication is a form of primary-backup fault tolerance; i.e. it tolerates the failure of any number of surrogates. For each application, Slingshot creates a first-class replica on a reliable server known to the mobile user—this server is referred to as the *home server*. Slingshot ensures that all application state can be reconstructed from information stored on the client and the home server. This allows all state on a surrogate to be regarded as soft state. Even if all surrogates crash, Slingshot continues to service requests using the first-class replica on the home server. In contrast, a naive approach that migrates applications between surrogates might lose state when a surrogate fails.

We note that Slingshot handles both *stateful* and *stateless* applications. The result of a remote operation for a stateful application depends upon the operations that have previously executed. Slingshot assumes that applications are deterministic; i.e. that given two replicas in the same initial state, an identical sequence of operations sent to each replica will produce identical results. As we discuss in Section 4.2, Slingshot adopts an approach similar to that of Rodrigues' BASE [24] in eliminating nondeterminism with wrapper code. Slingshot instantiates a new replica by checkpointing the first-class replica, shipping its volatile state to a surrogate, and replaying any operations that occurred after the checkpoint.

Instantiation of a new replica takes several minutes since the volatile state must travel through the bandwidth-constrained backhaul connection. However, existing replicas mitigate the perceived performance impact. Until the new replica is instantiated, existing replicas service application requests.

2.3 Ease of maintenance

We see the business case for deploying a surrogate as being similar to that of deploying a wireless access point. Desktop computers (without monitors) cost only a few hundred dollars today, not much more than an access point. Further, our results show that surrogates can provide significant value-add to wireless customers in terms of improved interactive performance.

However, surrogates must be easy to manage if they are to be widely deployed. Since we envision surrogates at hotspots in airport lounges, coffee shops, and bookstores, they must require little to no supervision. They should be appliances that require little configuration; most problems should be fixable with a reboot.

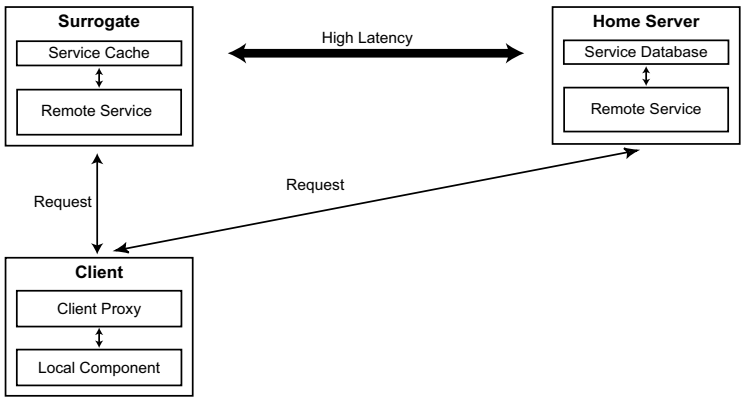


Figure 1. Slingshot architecture

To make surrogates easy to manage, Slingshot:

- **minimizes the surrogate computing base.** Each replica runs within its own virtual machine, which encapsulates all-application specific state such as a guest OS, shared libraries, executables, and data files. The surrogate computing base consists of only the host operating system (Linux), the virtual machine monitor (VMware), and Slingshot. No configuration or setup is needed to enable a surrogate to run new applications—each VM is self-contained.
- **uses a heavyweight virtual machine**. While para-virtualization and other lightweight approaches to virtualization offer scalability and performance benefits [4, 23, 29], they also restrict the type of applications that can run within a VM. In contrast, by using a heavyweight VMM (VMware), Slingshot runs the two applications described in Section 4 without modifying source code, even though their guest OS (Windows XP) differs substantially from the surrogate host OS (Linux).
- **places no hard state on surrogates.** Because surrogates have only soft state, a reboot does not lead to incorrect application behavior or data loss. If a surrogate crashes or is rebooted, the only impact a user sees is that performance declines to the level that would have been available had the surrogate never been present.

3 Slingshot implementation

3.1 Overview

Figure 1 shows an overview of Slingshot. For simplicity of exposition, this figure assumes that the mobile client is executing a single application and that a single surrogate is being used. In practice, we expect a Slingshot user to run only one or two applications concurrently, with each service replicated two or three times.

Each Slingshot application is partitioned into a *local component* that runs on the mobile client and a *remote service* that is replicated on the home server and surrogates. Ideally, we partition the application so that resource-intensive functionality executes as part of the remote service; the local component typically contains only the user interface. This partitioning enables demanding applications to run on clients such as handhelds that are highly portable but also resource-impoverished. The applications that we have studied so far (speech recognition and remote desktops) already had client-server partitionings that fit this model. For some applications, the best partitioning may not be immediately clear—in these cases, we could leverage prior work [2, 12, 18] to choose a partition that fits our model.

In Figure 1, a first-class replica executes on the home server and a second-class replica executes on the surrogate. The home server, described in Section 3.2, is a well-maintained server under the administrative control of the user, e.g. the user's desktop or a shared server maintained by the user's IT department. In contrast, surrogate computers, described in section 3.3, are co-located with wireless access points. They are administered by third parties and are not assumed to be reliable.

Slingshot creates the first-class replica when the user starts the application—this replica is required for execution of stateful services. As the application runs, Slingshot dynamically instantiates one or more second-class replicas on nearby surrogates. These replicas improve interactive performance because they are located closer to the user and respond faster than the first-class replica on the distant home server. If no second-class replicas are instantiated, Slingshot's behavior is identical to that of remote execution.

Each replica executes within its own virtual machine. Replica state consists of the *persistent state*, or disk im-

age of the virtual machine, and the *volatile state*, which includes its memory image and registers. To handle persistent state, we use the Fauxide and Vulpes modules developed by Intel Research's Internet Suspend/Resume (ISR) project [20]. These modules intercept VMware disk I/O requests. On the home server, we redirect these requests to a *service database* that stores the disk blocks of every remote service. On a surrogate, VMware reads are first directed to a *service cache*—if the block is not found in the cache, it is fetched from the service database on the home server.

The client proxy is responsible for locating surrogates, instantiating second-class replicas, and managing communication with all replicas. It presents the local component with the illusion that it is using a single remote service by broadcasting each request to all replicas and forwarding the first reply it receives to the local component. Later replies from other replicas are checked for consistency, as described in Section 3.4.

If a mobile computer has a high-capacity storage device such as a flash card or a microdrive, Slingshot reduces the time to instantiate replicas by storing checkpoints on the mobile computer. As described in Section 3.6, the client logs all operations that occur after the checkpoint and replays them to bring a new surrogate up-to-date.

3.2 The home server

A Slingshot user defines a single, well-known home server that stores and executes the first-class replicas for all of her remote services. Each service is uniquely identified by a *serviceid* string assigned by the user when the service is created. The service database on the home server manages the persistent and volatile state associated with each service. The *director* instantiates and terminates first-class replicas. We describe these components in the next two sections. The home server also runs the VMware virtual machine monitor. Each Slingshot service runs within a separate VM that is dynamically instantiated when a user starts its associated application on the client.

3.2.1 The service database

The home server stores the state of every service under its purview in its service database. Previous research in virtual machine migration by Sapuntzakis [25] and Tolia [26] has shown that content-addressable storage is highly effective in reducing the storage costs of virtual machine disk images. We adopt their approach by dividing the disk image of each virtual machine into 4 KB chunks and indexing each chunk by its SHA-1 hash value. As shown in Figure 2, each service has a *chunk table* that maps the chunks in its virtual disk image to the SHA-1 hash of the data stored at each location.

The service database assumes that any two blocks that hash to the same value are identical. It maintains a hash table of the SHA-1 values of all chunks that it currently stores. When it receives a request to store a new chunk whose SHA-1 value matches that of a chunk it already has stored, it increments a reference count on the existing chunk. This method of eliminating duplicate storage has been shown to substantially reduce disk usage [10] due to similarities between the disk state of different computers. We expect such similarities to be common in our environment, since a single user may create many remote services from a generic OS image. For example, we created the speech recognizer and VNC services discussed in Section 4 from the same Windows XP image.

As shown in Figure 2, when a first-class replica reads a block from its virtual disk, the Fauxide/Vulpes ISR modules intercept the request and pass the associated logical block number to the service database. The database looks up the block number in the service's chunk table to determine the SHA-1 value of the chunk stored at that location. It then looks up the SHA-1 value in the hash table to find the location of the chunk in the database.

Requests that modify blocks follow a similar path. The database locates the chunk in the service's chunk table. It then indexes on the old SHA-1 value and decrements the reference count associated with the chunk in its hash table. If the reference count drops to zero, it deletes the chunk. The service database next looks up the new SHA-1 value of the modified block in its hash table. If the modified chunk is a duplicate of an existing chunk, the service database increments the reference count of the existing chunk. Otherwise, it stores the chunk and adds its SHA-1 value to its hash table.

Since the volatile state is likely unique to each service, content-addressable storage offers little benefit. Thus, the service database stores the volatile state of each remote service in a file named by its serviceid.

3.2.2 The home server director

When a mobile user starts a Slingshot application, the client proxy asks the director on the home server to instantiate the first-class replica. The director uses the serviceid provided by the client proxy to retrieve the volatile state from the service database. It starts a VMware process, resumes the virtual machine with the volatile state, and replies to the client proxy. The persistent state is retrieved on demand from the service database as the first-class replica executes.

When the user terminates the application, the client proxy tells the director to halt the replica. The director suspends the virtual machine, which causes VMware to write its volatile state to disk. It then terminates the virtual machine.

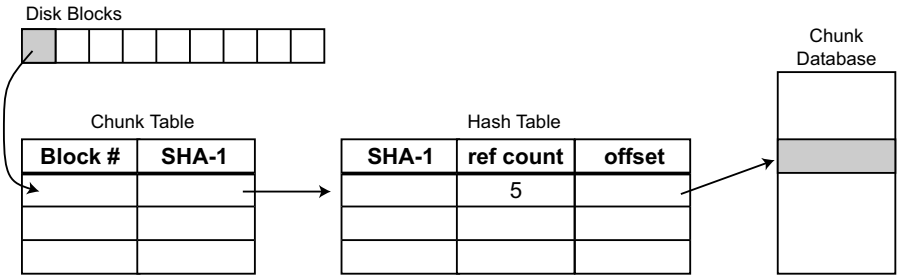


Figure 2. Reading a chunk from the service database

The volatile state is large (e.g. 128 MB) because it contains the entire memory image of the virtual machine. We use Waldspurger's *ballooning* technique [28] to reduce its size. When we create a new service, we place a script in the guest OS that allocates a large amount of highly-compressible (e.g. almost entirely zeroed out) memory pages. When VMware suspends the virtual machine, this script runs to force unused memory pages to disk and replace them with more compressible pages. The director compresses the volatile state with gzip before storing it in the service database—this reduces storage and network costs.

3.3 Surrogates

The surrogate architecture is similar to that of the home server, except that we replace the service database with the service cache described in Section 3.3.2. The director also plays a slightly different role on a surrogate.

3.3.1 The surrogate director

The surrogate director currently accepts connections from any client that wishes to instantiate a second-class replica. Potentially, the director could enforce access-control policies similar to those enforced by access points today. The client proxy passes the director the IP address of its home server and the serviceid of the remote service it wishes to instantiate. The director contacts the home server and requests the volatile state and chunk table for the requested service.

Usually, the home server is already executing the first-class replica of the service in question. For a stateful service, this means that Slingshot must generate a coherent checkpoint that represents the current execution state of that replica. The home server creates this checkpoint by suspending and resuming the virtual machine containing the replica; this causes a new volatile state and chunk table to be written to the service database.

The home servers ships copies of the volatile state and chunk table to the surrogate. Even after compression, this information is quite large (e.g. 32 MB for a VNC service)—thus, it can take several minutes to transfer. Next, the director starts a new virtual machine and resumes it using the volatile state. As the replica executes, its disk I/O is intercepted by the ISR modules and redirected to the service cache described below.

When the client disconnects from the surrogate, the director terminates the virtual machine. Since surrogate replicas are second-class, the service state is logically discarded at this point. However, persistent state chunks remain in the service cache until they are evicted due to storage limitations. This improves response time if the service is later restarted on the surrogate.

3.3.2 The service cache

The service cache is a content-addressable store of data chunks. As with the service database, each chunk is indexed by its SHA-1 hash value, and storage of duplicate chunks is eliminated. This lets users benefit from similarities among the disk images of their replicas. For instance, two people using Windows-based services may have similar disk images. Chunks cached by one user can be used by the other.

When the service cache receives a request to read a chunk, it first tries to service it locally. If the chunk is not cached, it asks the service database associated with the replica for the chunk.

A subtle problem occurs because Slingshot enforces determinism at the application level. There is no guarantee that two different replicas will write the same data to the same location on disk. A naive implementation might ask the database for an uncached chunk, only to find that it had been over-written by a store performed by the first-class replica. We therefore need to ensure that the service database keeps all chunks that might potentially be requested by second-class replicas.

Slingshot uses a copy-on-write approach for stateful services. When a surrogate starts a second-class replica, the database copies the service's chunk table—the new copy increments the reference count for each of its entries. When the second-class replica is terminated, the database deletes its chunk table and decrements the reference count for each entry. Thus, even if the first-class

replica modifies or deletes a chunk, that chunk is not deleted until after the second-class replica terminates.

A similar concern arises for chunks modified by the second-class replica. The modified chunks may not be written to the service database by the first-class replica due to non-determinism at the disk I/O level. The surrogate cache therefore pins modified chunks for the duration of a replica's execution—this ensures that they will never need to be fetched from the service database.

The surrogate cache uses an LRU eviction algorithm that exempts chunks that are currently pinned. Since chunks remain cached even after a service is terminated, it is likely that the chunks of a frequent visitor to a hotspot will remain cached between visits.

3.4 The client proxy

The client proxy is a stand-alone process responsible for surrogate discovery, instantiating and destroying replicas, and coordinating communication with each replica. It uses UPnP [22] to discover new surrogates in its surrounding network environment. Currently, the decision to instantiate a new second-class replica is a made by the user. In the future, we plan to add heuristics for monitoring network performance and automatically deciding when new replicas are needed.

On startup, the local component sends the client proxy its serviceid. The proxy immediately instantiates a first-class replica on the home server. It subsequently instantiates second-class replicas on nearby surrogates when requested by the user.

The client proxy maintains an *event log* of requests sent by the local application component. The client proxy spawns a thread for each replica; the thread sends logged events to the replica in the order they were received. Events may optionally have application-specific preconditions that must be satisfied before they can be sent to a replica. For instance, our VNC application specifies a precondition that ensures that the remote desktop is ready to accept each key stroke and mouse click event before that event is sent. Services that must process events sequentially to ensure determinism specify that the previous event must complete before an event is sent.

The client proxy records the replies received from each replica in the event log. When the first reply is received, it is returned to the local component. Later replies are compared with the first reply to ensure that the replicas are behaving deterministically. If the reply from a second-class replica differs significantly (as determined by an application-specific function) from the reply from the first-class replica, the second-class replica is terminated. Such divergence could be due to a bug in the wrapper code enforcing determinism, or it could be the result of a faulty or malicious surrogate. Note that the client proxy may already have received a reply that is later determined to be faulty. In this case, the application is notified via an upcall so that corrective action can be taken. This strategy is similar to those employed in the SUNDR file system [21] and in Brown's operator undo [7]. Alternatively, we could try to prevent malicious surrogate behavior using a trusted computing architecture [15].

3.5 Instantiating new replicas

In Figure 3, we show how Slingshot responds to a user moving between hotspots, assuming that a surrogate is located at each hotspot. The client proxy first asks the nearby surrogate to instantiate a replica. The surrogate requests the service state from the home server; the home server checkpoints the first-class replica and ships the compressed chunk table and volatile state to the surrogate. Note that the distant surrogate can process events for the client during checkpointing. This hides almost all delay associated with suspending and resuming the VM on the home server. The client proxy queues events for the first-class replica while it is being checkpointed and sends them after the replica resumes execution.

The new surrogate uses the checkpoint to resume the service within a new virtual machine. The client proxy brings the new replica up-to-date by replaying all events in the event log that were sent by the application after the checkpoint was created. Once the new replica is up-to-date, it improves interactive response time for the application by responding more swiftly to new events sent by the local component. At this point, the client proxy terminates the replica at the previous hotspot.

The benefit of replication is that the user sees little foreground performance impact due to the use of a new surrogate. After checkpointing, the first-class replica on the home server services requests while the new second-class replica is instantiated and brought up-to-date. In contrast, a naive migration approach would leave the service unavailable while state is being shipped—as we show in Section 5, application state can take several minutes to ship over limited backhaul connections. Although the first-class replica is unavailable while it is being checkpointed, that operation is relatively short (i.e. approximately 10 seconds). Even that cost can be masked if another second-class replica exists.

Slingshot performs two optimizations if a service is marked as stateless. It skips checkpointing the service on the home server (since its state is static). It also does not replay events (since the replica is up-to-date).

3.6 Leveraging mobile storage

Migration can be time consuming due to the need to ship state from the home server (step 4 in Figure 3). For a typ-

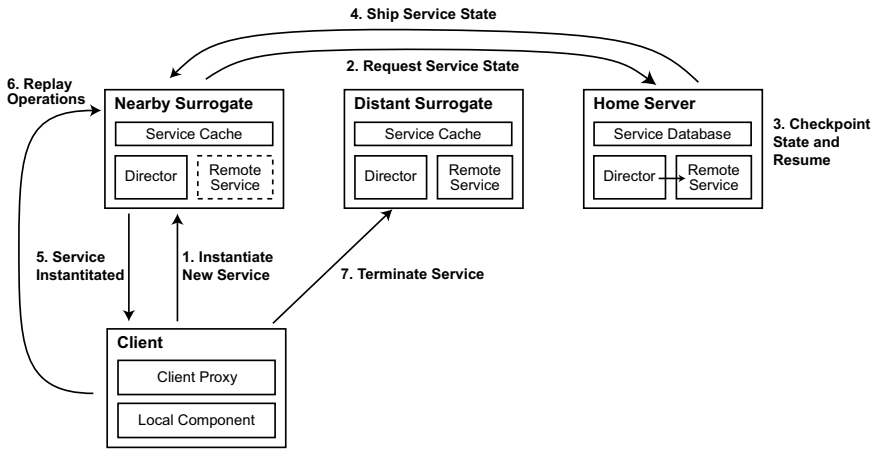


Figure 3. Instantiating a new replica

ical service, the size of the compressed volatile state and chunk table is 30–40 MB. If the home server is connected to the Internet via a DSL link with 256 Kb/s uplink bandwidth, it takes over 20 minutes to ship the state.

Given sufficient storage capacity, Slingshot reduces the time to ship state by storing checkpoints on the mobile computer. We observed that the service-specific event log can be used to roll forward replica state from *any* prior checkpoint, not just one that is created at the beginning of replica instantiation. Thus, by storing a checkpoint on a mobile computer and logging all events that occur after that checkpoint, Slingshot can instantiate a replica without shipping state from the home server. Instead, it ships the state from the mobile computer over the high-bandwidth wireless network at the hotspot. Most of this bandwidth should be unused since the capacity of the wireless network is typically much greater than that of the backhaul connection, yet most communication from computers located at the hotspot is with endpoints located outside the hotspot (and thus limited by the backhaul bandwidth).

When a user returns to her home server, she can tell Slingshot to create new checkpoints of her applications on a high-capacity storage device such as a microdrive. Each checkpoint contains the volatile and persistent state. The volatile state and chunk table are stored in separate files; the chunks that comprise the persistent state are stored in a content-addressable cache on the mobile computer. The event log is empty when the user creates a new snapshot. As the application is used on the road, Slingshot appends each request to the log. This enables Slingshot to instantiate a new replica of a stateful service by first restoring the checkpoint represented by the volatile state, and then deterministically replaying the event log. For stateless services, Slingshot neither records nor replays an event log.

When a new replica is instantiated on a nearby surrogate, the mobile computer tries to find a checkpoint on its local storage device. If a checkpoint is found, the mobile computer ships the volatile state, chunk table, and hash table for its local chunk cache to the surrogate. One reason that we transmit the chunk table and the hash table to the surrogate is that the surrogate can usually maintain this information in memory, whereas a resource-constrained mobile computer cannot. When a disk I/O request misses in the service cache, the surrogate fetches the chunk from the mobile computer if it is available there; otherwise, the chunk is fetched from the home server.

As operations accumulate, so does the time to bring a new second-class replica up-to-date. This means that there exists a break-even point where it takes less time to create a new checkpoint from the first-class replica on the home server and download it over the Internet than it takes to instantiate a replica from client storage.

4 Slingshot applications

We have adapted the IBM ViaVoice speech recognizer and the VNC remote desktop to use Slingshot. Due to Slingshot's use of virtual machine encapsulation, we did not need to modify the source code of either application. All Slingshot-specific functionality is performed within proxies that intercept and redirect network traffic.

4.1 Speech recognition

We chose speech recognition as our first service because of its natural application to handheld computers. We used IBM ViaVoice in our work. We created a server-side proxy that accepts audio input from a remote client and passes it to ViaVoice through that application's API. ViaVoice returns a text string which the proxy sends to the client. ViaVoice and our server run on a Windows XP guest OS executing within a VMware virtual ma-

chine. The local component of this application displays the speech recognition output.

We chose to implement speech recognition as a stateless service. One can certainly make a reasonable argument that speech recognition should be a stateful service in order to allow a user to train the recognizer. However, we wanted to explore the optimizations that Slingshot could provide for stateless services.

4.2 Virtual desktop

VNC allows users to view and interact with another computer from a mobile device. In the case of Slingshot, the remote desktop is a Windows XP guest OS executing within a VMware virtual machine. This allows users to remotely execute any Windows application from their handhelds. This is clearly a stateful service; i.e., a user who edits a Word document expects the document to exist when the service is next instantiated.

Adapting VNC to Slingshot presented interesting challenges. First, the VNC server sends display updates to the client in a non-deterministic fashion. When pixels on the screen change, it reports the new values to the client in a series of updates. Two identical replicas may communicate the same change with a different sequence of updates. The resulting screen image at the end of the updates is identical but the intermediary states may not be equivalent. A second challenge is that some applications are inherently non-deterministic. One annoying example is the Windows system clock; two surrogates can send different updates because their clocks differ.

We noted that some non-determinism is unlikely to be relevant to the user (e.g. a slightly different clock value). Unfortunately, other non-determinism affects correct execution. For example, a key stroke or mouse click is often dependent upon the window state. If a user opens a text editor and enters some text, the key strokes must be sent to each replica only after the editor has opened on that replica. If this is not done, the key strokes will be sent to another application. To solve this problem, we associate a precondition with each input event. When the user executes the event, we log the state of the window on the client to which that event was delivered. When replaying the event on a server, we require that the window be in an identical state before the event is delivered. Since each event is associated with a screen coordinate, we check state equality by comparing the surrounding pixel values of the original execution and the background execution. In the above example, this strategy causes Slingshot to wait until the editor is displayed before it delivers the text entry events.

A second issue with VNC is that its non-determinism prevents us from mixing updates from different replicas. We designate the best-performing replica as the foreground replica and the remainder as background replicas. Only events from the foreground replica are delivered to the client. If performance changes, we quiesce the replicas before choosing a new foreground replica. Two replicas are quiesced by ensuring that the same events have been delivered to each, and by requesting a full-screen update from the new foreground replica to eliminate transition artifacts. New events are logged while quiescing replicas. Note that the foreground replica is rarely the first-class replica since nearby surrogates provide better performance in the common case.

We were encouraged that VNC can fit within the Slingshot model, since its behavior is relatively non-deterministic. Based on this result, we suspect that application-specific wrappers can be used to enforce determinism for many applications. For those applications where this approach proves infeasible, we could use a VMM that enforces determinism at the ISA level as is done in Hypervisor [6] and ReVirt [11].

5 Evaluation

Our evaluation answers the following questions:

- How much do surrogates improve interactive response time?
- What is the perceived performance impact of instantiating a new replica?
- How much does the use of mobile storage reduce replica instantiation time?

5.1 Methodology

The client platform in our evaluation is an iPAQ 3970 handheld running the Linux 2.4.19-rmk6 kernel. The handheld has an XScale-PXA250 processor, 64 MB of DRAM, and 48 MB of flash. It uses a 11 Mb/s Cisco 350 802.11b PCMCIA card for network communication and a 4 GB Hitachi microdrive for bulk storage. Unless otherwise noted, the home server and surrogates are Dell Precision 350 desktops with a 3.06 GHz Pentium 4 processor running RedHat Enterprise Linux version 3.

We use a Cisco 350 802.11b wireless access point. We emulate the topology in Figure 4 by connecting all computers and the access point to a Dell desktop running the NISTNet [8] network emulator. This topology emulates a scenario where the handheld client is located at a wireless hotspot equipped with a surrogate computer. Hotspots are connected to the Internet through T1 connections with 1.5 Mb/s uplink and downlink bandwidth. A distant surrogate at another hotspot is accessible with latency of 15 ms. The home server is connected through

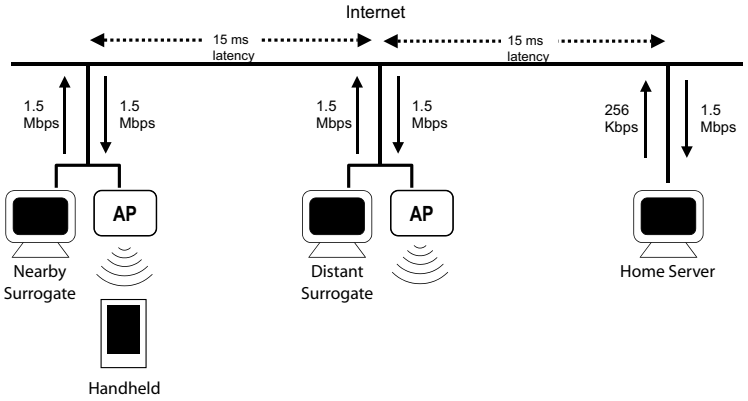


Figure 4. Network topology used in evaluation

an emulated DSL connection—the latency between the handheld's hotspot and the home server is 30 ms.

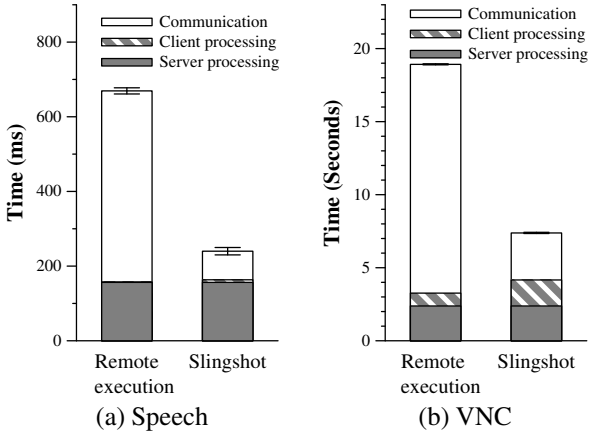
We execute the IBM ViaVoice speech recognizer as a stateless service, and VNC as a stateful service. For repeatability, the local component of each application executes a fixed, periodic workload. For speech, each iteration of the workload recognizes a phrase and pauses three seconds before beginning the next iteration. For VNC, each iteration opens Microsoft Word, inserts text at the beginning of a document, saves the document, closes Word, and pauses ten seconds before the next iteration begins. The client uses the same heuristics described in Section 4.2 to wait until Word opens before inserting text, and to wait until the window is fully closed before beginning the 10 second pause between iterations.

Each service runs within a separate VM configured with 128 MB of memory and 4 GB of local storage. We create each service from a vanilla Windows XP installation. We install the ballooning script described in Section 3.2.2 and the application comprising the remote service. We start the application so that it is ready to receive incoming connections, then suspend the VM. We repeat each experiment three times and report mean results over all iterations during the three trials. Figures 12 and 13 summarize all results described in this section.

5.2 Benefit of Slingshot

We first measured the benefit of using Slingshot for our two applications. The left bar in each data set in Figure 5 shows the average time to perform an iteration of the workload when the service is remotely executed on the home server. The right bar shows the average time using Slingshot when a second-class replica executes on the nearby surrogate. We let each application run for several iterations before measuring performance; this eliminates startup transients and shows steady-state performance.

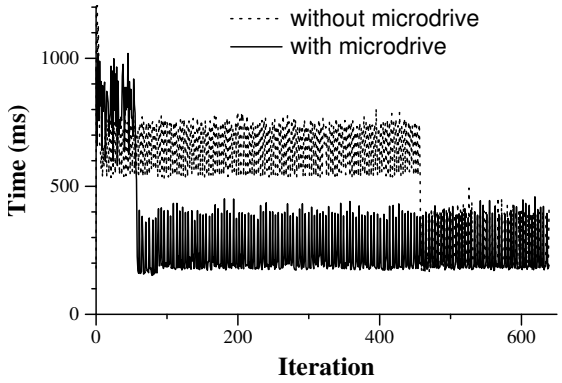
Both the stateless speech service and the stateful VNC service execute 2.6 times faster with Slingshot than with remote execution. The shadings within each bar show the time consumed by server processing, client processing, and communication. For speech, Slingshot increases client processing time since it manages multiple network connections and aggregates responses. For VNC, it also logs requests and responses to local storage. Slingshot's performance benefit comes from reducing the time the application blocks on network communication.



This graph compares the average time to execute the speech and VNC workloads when using remote execution and when using Slingshot. Each bar shows mean response time—the error bars are 90% confidence intervals.

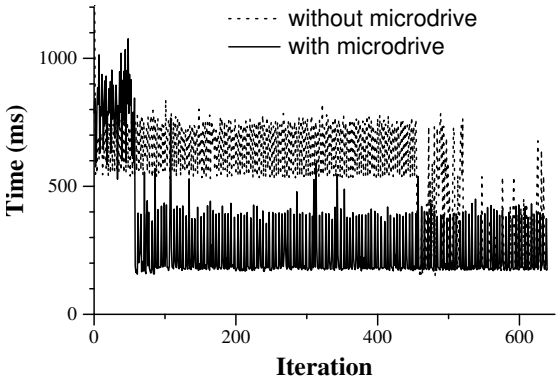
Figure 5. Benefit of using Slingshot

Remote execution performance is affected by both high latency and limited bandwidth. For speech, a back-of-the-envelope calculation shows that 229 ms are required to transfer the 44 KB utterance through the bottleneck 1.5 Mb/s T1 link at the hotspot. Further, since communication is intermittent, TCP slow start causes several 60 ms round-trip delays during transmission. Thus, the remote execution results include 511 ms of network communication time. In contrast, Slingshot uses only 77 ms for network communication.



This graph shows how response time changes during the instantiation of a speech replica on the nearby surrogate. All chunks are in the service cache prior to each experiment.

Figure 6. Speech replication with warm cache



This graph shows how response time changes during the instantiation of a speech replica on the nearby surrogate. No chunks are in the service cache prior to each experiment.

Figure 7. Speech replication with cold cache

Latency impacts VNC performance more than bandwidth. Because the client waits for remote actions such as button clicks and key presses to complete before initiating the next action, there are many round-trip delays during the VNC workload. In addition, client polling in VNC leads to more round-trip delays than are strictly necessary. For this workload, remote execution on the home server requires 15.6 seconds for network communication, while Slingshot requires only 3.2 seconds.

5.3 Stateless service replication

We next examined the impact of instantiating stateless second-class replicas. In this experiment, a user with a first-class replica running on the home server arrives at the hotspot on the left in Figure 4 and decides to instantiate a replica on the surrogate there. For repeatability, we do not measure the latency of UPnP service discovery. We examine behavior when the service cache is cold (i.e. no chunks are initially cached) and warm (i.e. all chunks are initially cached). The warm cache scenario is most likely if the user has recently visited the hotspot; the cold cache scenario is the worst cache state possible.

We first ran three trials without a microdrive attached to the iPAQ. Since the handheld has limited storage, the service state must be loaded from the home server as described in Section 3.5. We then ran three trials with the microdrive; in this case, the state of the speech service is loaded from the iPAQ as described in Section 3.6. Figures 6 and 7 show results for representative trials with a warm and cold cache, respectively.

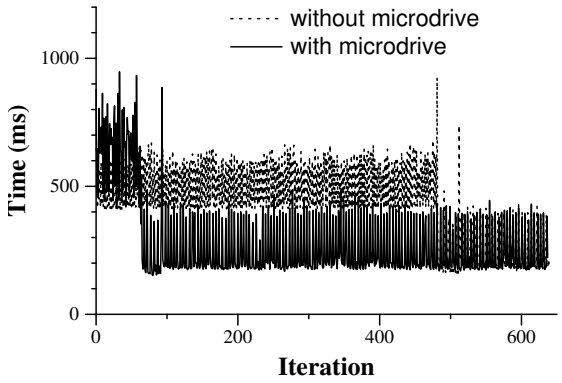
In Figure 6, the sharp drop in response time for both lines is a result of the completion of replica instantiation. Before the replica is instantiated, speech requests must be serviced by the distant home server; after instantiation, the new second-class replica provides quicker response time. Without the microdrive, it takes 28:06 minutes to ship the service state from the home server. However, replica instantiation exhibits only a minimal impact on application performance—average response time during replication is only 2% greater than response time with remote execution on the home server.

When the replica is instantiated from state stored on the client's microdrive, the new second-class replica is instantiated in only 3:35 minutes (7.8 times faster). However, the performance impact of replica instantiation is more substantial: average response time increases by 20% compared to remote execution. Shipping a large amount of data over the wireless network causes queuing delays at the access point and on the handheld that adversely affect application performance. Currently, we are investigating whether traffic prioritization can minimize the impact of replication on foreground traffic.

The cold cache scenario in Figure 7 exhibits a less clear difference in performance before and after replication completes. After the new replica is instantiated, it fetches chunks of its persistent state on demand from the home server or iPAQ; this occasionally delays its responses. Note that the first-class replica on the home server mitigates the performance impact—if the second-class replica is substantially delayed by fetching state, the first-class replica responds faster.

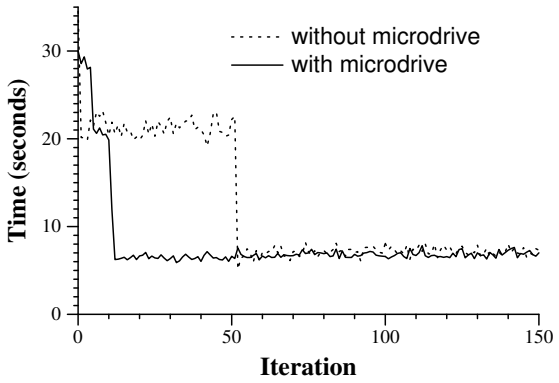
5.4 Instantiation of another stateless replica

We next examined a scenario in which the user of the speech service moves from one wireless hotspot to another. This experiment begins with the user located at the middle hotspot in Figure 4. A second-class replica is running on the surrogate at that hotspot and a first-class replica is running on the home server. At the beginning of the experiment, the user moves to the left hotspot and decides to instantiate another replica on the surrogate at that hotspot. While this new replica is being created,



This graph shows how response time changes during the instantiation of a speech replica on the nearby surrogate while another replica executes on the distant surrogate. All chunks are in the the service caches prior to each experiment.

Figure 8. Speech: Moving to a new hotspot



This graph shows how response time changes during the instantiation of a VNC replica on the nearby surrogate. All chunks are in the service cache prior to each experiment.

Figure 9. VNC replication with warm cache

both the second-class replica on the distant surrogate and the first-class replica on the home server service application requests. As soon as the new replica is instantiated, Slingshot terminates the replica on the distant surrogate. Since we did not have three identical servers with which to run this experiment, the home server is a slightly less-powerful Dell Optiplex 370 with 2.8 GHz Pentium 4 processor running RedHat 9.

Figure 8 shows how the average time to perform an iteration of the speech recognition workload varies during this experiment—we show only warm cache results here. Compared to the previous experiment, the time to instantiate a new replica is relatively unchanged. However, response time during replication is improved because the existing second-class replica responds faster to requests than the replica on the home server. Without the microdrive, application response time is reduced by 23% compared to remote execution; with the microdrive, application response time is reduced by 2%. These results show that a surrogate can still provide significant benefit even when not located at the user's current hotspot.

5.5 Stateful service replication

We next repeated the experiment in Section 5.3 for the stateful VNC service. Prior to the experiment, we perform 30 iterations of the VNC workload. We then begin the experiment by instantiating a replica on the nearby surrogate. Figures 9 and 10 show results from the warm and cold cache scenarios, respectively.

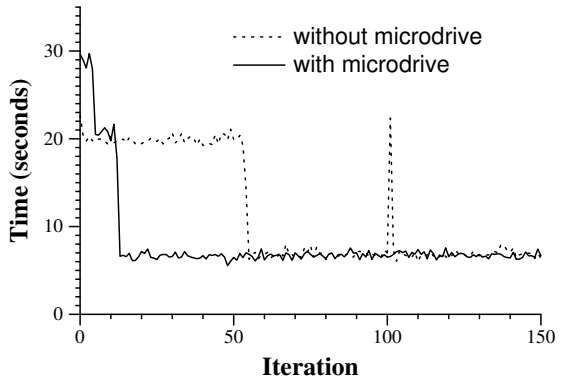
Without the microdrive and with a warm service cache, Slingshot takes 4 seconds to checkpoint the VNC service—this is reflected in the higher response time for the first iteration. Slingshot takes 22:42 minutes to ship the checkpoint to the surrogate and 5:02 to replay the logged operations. During replication, average response time is 20% higher than when using remote execution on the home server. This increase is due to the background traffic associated with shipping state from the home server interfering with the latency-sensitive foreground traffic of VNC.

In contrast to the prior results for the speech service, the VNC results show little difference between the warm and cold cache scenarios. In particular, VNC performance markedly improves in the cold cache scenario as soon as the client starts using the second-class replica. Most of the chunks needed by the service are read from the service database during the replay of logged operations.

When the handheld stores a VNC service checkpoint on its microdrive, Slingshot takes 3:19 minutes to ship the state from the client, and 3:18 minutes to replay the log. These two phases are clearly visible in the "with microdrive" line in Figure 9, where response time degrades by 52% compared to remote execution while state is being shipped, and by 9% while the log is replayed. Note that the log replay with the microdrive includes the 30 iterations that occurred prior to the experiment. Since the microdrive checkpoint is taken when the user leaves home, all logged operations after that point must be replayed. However, since shipping state takes less time with the microdrive, the user generates fewer logged operations during migration. Overall, Slingshot instantiates the replica over 4 times faster when a checkpoint exists on the microdrive.

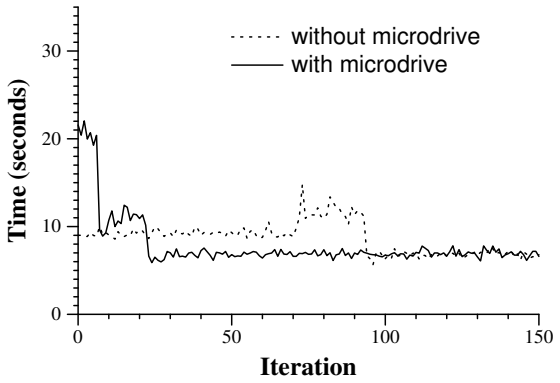
5.6 Instantiation of another stateful replica

We also repeated the experiment in Section 5.4 for VNC. Prior to the experiment, we create a second-class replica of the VNC service on the distant surrogate and a first-class replica on the home server. We then execute 30 iterations of the VNC workload. The experiment begins



This graph shows how response time changes during the instantiation of a VNC replica on the nearby surrogate. No chunks are in the surrogate cache prior to each experiment.

Figure 10. VNC replication with cold cache



This graph shows how response time changes during the instantiation of a VNC replica on the nearby surrogate while another replica executes on the distant surrogate. All chunks are in the the service caches prior to each experiment.

Figure 11. VNC: Moving to a new hotspot

when we start to instantiate another second-class replica on the nearby surrogate.

As shown in Figure 11, the presence of another second-class replica on the distant surrogate substantially improves performance during replication. Compared to remote execution, Slingshot provides VNC response times almost twice as fast without the microdrive, and 70% faster when state is fetched from client storage.

6 Related work

To the best of our knowledge, Slingshot is the first system to dynamically instantiate replicas of stateful applications in order to improve the performance of small, resource-poor mobile computers. Our work draws upon several areas of prior work, including virtual machine and process migration, cyber foraging, fault-tolerant computing, and remote execution.

After Chen and Noble [9] first suggested that virtual machine migration could be an effective mechanism for process migration, several research groups have built working prototypes. Our research focus is not on the migration mechanism itself, but rather on how it can be used to service the needs of small, mobile clients. We use the Fauxide and Vulpes components from Intel's Internet Suspend/Resume [20] to intercept disk I/O requests made by virtual machines. The difference between ISR and Slingshot is that ISR executes a user's computing environment on a single terminal at a time. In contrast, Slingshot decomposes a user's environment into distinct services and replicates services on multiple computers. Slingshot hides the perceived latency of migration and surrogate failures, while letting a user execute applications anywhere a wireless connection exists.

Sapuntzakis [25] uses virtual machine migration, but focuses on users who compute at fixed locations, rather than the mobile users that Slingshot targets. Slingshot uses several of the optimizations suggested by Sapuntzakis, including ballooning and content-addressable storage. These optimizations have also been suggested by Waldspurger [28] and Tolia [27], respectively.

Baratto's MobiDesk [3] is similar to our VNC application in that it virtualizes a remote desktop for mobile clients. However, MobiDesk migrates its desktop service between well-connected machines in a cluster in order to minimize downtime during server maintenance or upgrades. Slingshot uses replication rather than migration, and utilizes the computational resources of surrogates located at wireless hotspots. A MobiDesk-like cluster could serve as the ideal home server for Slingshot applications. Conversely, although Baratto shows considerable improvement over VNC in remote display performance, his results indicate that network latency still degrades interactive performance. Thus, surrogates could improve MobiDesk performance for mobile clients.

Slingshot's replication strategy is a form of primary-backup fault tolerance in that the replica on the home server allows the system to tolerate a fail-stop failure of any number of second-class replicas. Our approach is most reminiscent of Hypervisor [6], which used deterministic replay to provide fault tolerance between virtual machines. In contrast to systems such as Hypervisor and ReVirt [11] which enforce determinism at the ISA level, Slingshot enforces determinism at the application level. This choice was driven by our desire to use a robust commercial virtual machine (VMware) without modification. Our approach to enforcing determinism was inspired by Rodrigues' BASE [24], which provides Byzantine fault tolerance by wrapping non-deterministic software with a layer that enforces deterministic behavior. A similar approach was used in Brown's operator undo [7].

Slingshot is an instance of cyber foraging [1], the oppor-

Service	Remote Execution	Slingshot				
		Steady-State	Creating 1st Replica		Creating 2nd Replica	
			Warm Cache	Cold Cache	Warm Cache	Cold Cache
Speech w/o microdrive	0.67 (0.67–0.67)	0.24 (0.24–0.24)	0.69 (0.68–0.70)	0.69 (0.67–0.73)	0.52 (0.51–0.52)	0.52 (0.51–0.52)
Speech with microdrive	0.67 (0.67–0.67)	0.24 (0.24–0.24)	0.80 (0.79–0.81)	0.80 (0.79–0.81)	0.65 (0.65–0.65)	0.65 (0.63–0.68)
VNC w/o microdrive	18.9 (18.9–19.0)	7.4 (7.2–7.5)	22.8 (21.5–23.6)	22.2 (19.9–23.6)	9.8 (9.7–9.9)	10.0 (9.9–10.0)
VNC with microdrive	18.9 (18.9–19.0)	7.4 (7.2–7.5)	24.1 (23.9–24.3)	23.1 (21.6–24.4)	13.8 (13.0–14.4)	14.6 (14.4–14.8)

This figure summarizes the average response time (in seconds) for all experiments. Each entry shows the mean of three trials, with the low and high trials given in parentheses. The second column shows response time for remote execution on the home server. The third column shows steady-state performance for Slingshot with a replica on the nearby surrogate. The remaining columns show response time while instantiating a replica on the nearby surrogate with and without a replica running on the distant surrogate.

Figure 12. Summary of response time results

Service	Creating 1st Replica		Creating 2nd Replica	
	Warm Cache	Cold Cache	Warm Cache	Cold Cache
Speech w/o microdrive	28:06 (27:50–28:27)	27:55 (27:50–28:04)	28:10 (28:05–28:11)	27:57 (27:57–27:58)
Speech with microdrive	3:35 (3:32–3:40)	3:27 (3:26–3:28)	3:39 (3:34–3:45)	3:33 (3:32–3:34)
VNC w/o microdrive	27:48 (27:07–28:28)	27:58 (27:12–28:45)	31:16 (30:57–31:31)	31:08 (31:00–31:25)
VNC with microdrive	6:37 (6:20–7:13)	7:29 (6:59–8:27)	8:59 (8:01–10:00)	8:20 (6:47–9:00)

This figure summarizes the time (in minutes) to create a new replica on the nearby surrogate for all experiments. Each entry shows the mean of three trials, with the low and high trials given in parentheses. The second and third columns show the time to instantiate a replica when no replica runs on the distant surrogate. The last two column show results with a replica on the distant surrogate.

Figure 13. Summary of replication time results

tunistic use of surrogates to augment the capabilities of mobile computers. Previous work in Spectra [12] examined how a cyber foraging system could locate the best server and application partitioning to use given dynamic resource constraints. In contrast, Slingshot takes this selection as a given and provides a mechanism for utilizing surrogate resources. More recently, Balan [2] and Goyal [16] have also proposed cyber foraging infrastructure. Compared to these systems, the major capability added by Slingshot is the ability to execute stateful services on surrogate computers. Data staging [13] and fluid replication [19] use surrogates to improve the performance of distributed file systems. They share common goals with Slingshot such as minimization of latency and ease of management—however, Slingshot applies these principles to computation rather than storage.

The applications we have investigated so far have been easy to partition because they were designed for client-server computing. Potentially, Slingshot could use one of several methods that automatically partition applications. For instance, Coign [18] partitions DCOM applications into client and server components. Globus [14], Condor [5], and Legion [17] dynamically place functionality, but target grid rather than mobile computing.

7 Conclusions and future work

Handhelds can improve interactive response time by leveraging surrogate computers located at wireless hotspots. Slingshot's use of replication offers several improvements over a strategy that simply migrates remote services between computers. Replication provides good response time for mobile users who move between wireless hotspots; while a new replica is being instantiated, other replicas continue to service user requests. Replication also lets Slingshot recover gracefully from surrogate failure, even when running stateful services.

Slingshot minimizes the cost of operating surrogates. For these computers to be of maximum benefit, they must be located at wireless hotspots, rather than in machine rooms that are under the supervision of trained operators. Slingshot uses off-the-shelf virtual machine software to eliminate the need to install custom operating systems, libraries, or applications to service mobile users. All application-specific state associated with each service is encapsulated within its virtual machine. Further, Slingshot's replication strategy means that surrogates need not provide 24/7 availability. If a surrogate fails or is rebooted, no state is lost.

Harnessing surrogate computation is a complex problem. Slingshot currently provides several pieces of the puzzle, including the use of replication to improving response time and the elimination of hard surrogate state to improve ease of management. Other pieces of the puzzle remain. Slingshot does not yet address the privacy issues inherent to running computation on third-party hardware. Trusted computing efforts [15] provide promise in this area. Slingshot does not provide a mechanism for securely controlling replica instantiation and termination. Other areas of potential investigation are load management and policies for creating and destroying replicas. We believe that Slingshot will be an extremely useful platform on which to conduct such investigations.

Acknowledgments

We are grateful to Mike Kozuch and the ISR team for providing us with the Fauxide and Vulpes components used in this work. We also thank Manish Anand, Edmund B. Nightingale, Daniel Peek, the anonymous reviewers, and our shepherd, Doug Terry, for comments that helped improve this paper. This research was supported by CAREER grant CNS-0346686 from the National Science Foundation and by an equipment grant from Intel Corporation. The views and conclusions in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, Intel, the University of Michigan, or the U.S. government.

References

[1] BALAN, R., FLINN, J., SATYANARAYANAN, M., SINNAMOHIDEEN, S., AND YANG, H.-I. The case for cyber foraging. In *the 10th ACM SIGOPS European Workshop* (Saint-Emilion, France, September 2002).

[2] BALAN, R. K., SATYANARAYANAN, M., PARK, S., AND OKOSHI, T. Tactics-based remote execution for mobile computing. In *Proceedings of the 1st Annual Conference on Mobile Computing Systems, Applications and Services* (San Francisco, CA, May 2003), pp. 273–286.

[3] BARATTO, R. A., POTTER, S., SU, G., AND NIEH, J. MobiDesk: Mobile virtual desktop computing. In *Proceedings of the 10th Annual Conference on Mobile Computing and Networking* (Philadelphia, PA, Sept/Oct 2004), pp. 1–16.

[4] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., AND HARRIS, T. Xen and the art of virtualization. In *Procedings of the 19th ACM Symp. on Operating Systems Principles* (Bolton Landing, NY, October 2003), pp. 164–177.

[5] BASNEY, J., AND LIVNY, M. Improving goodput by coscheduling CPU and network capacity. *International Journal of High Performance Computing Applications 13*, 3 (Fall 1999).

[6] BRESSOUD, T. C., AND SCHNEIDER, F. B. Hypervisor-based fault-tolerance. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)* (Copper Mountain, CO, December 1995), pp. 1–11.

[7] BROWN, A. B., AND PATTERSON, D. A. Rewind, repair, replay: Three R's to dependability. In *the 10th ACM SIGOPS European Workshop* (St. Emilion, France, September 2002).

[8] CARSON, M. *Adaptation and Protocol Testing thorugh Network Emulation*. NIST, http://snad.ncsl.nist.gov/itg/nistnet/slides/index.htm.

[9] CHEN, P., AND NOBLE, B. When Virtual is Better Than Real. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems* (Schloss Elmau, Germany, May 2001).

[10] COX, L. P., MURRAY, C. D., AND NOBLE, B. D. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (December 2002), pp. 285–298.

[11] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (December 2002), pp. 211–224.

[12] FLINN, J., NARAYANAN, D., AND SATYANARAYANAN, M. Self-tuned remote execution for pervasive computing. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)* (Schloss Elmau, Germany, May 2001), pp. 61–66.

[13] FLINN, J., SINNAMOHIDEEN, S., TOLIA, N., AND SATYANARAYANAN, M. Data staging for untrusted surrogates. In *Proceedings of the 2nd USENIX Conference on File and Storage Technology* (San Francisco, CA, March/April 2003), pp. 15–28.

[14] FOSTER, I., KESSELMAN, C., NICK, J., AND TUECKE, S. Grid services for distributed system integration. *Computer 35*, 6 (2002).

[15] GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., AND BONEH, D. Terra: A virtual machine-based platform for trusted computing. In *Procedings of the 19th ACM Symp. on Operating Systems Principles* (Bolton Landing, NY, October 2003), pp. 193–206.

[16] GOYAL, S., AND CARTER, J. A lightweight secure cyber foraging infrastructure for resource-constrained devices. In *Proceedings of the 6th IEEE Workshop on Mobile Computing Systems and Applications* (Lake Windermere, England, December 2004).

[17] GRIMSHAW, A. S., AND WULF, W. A. Legion: Flexible support for wide-area computing. In *Proceedings of the 7th ACM SIGOPS European Workshop* (1996).

[18] HUNT, G. C., AND SCOTT, M. L. The Coign automatic distributed partitioning system. In *Proceedings of the 3rd Symposium on Operating System Design and Implemetation (OSDI)* (New Orleans, LA, February 1999), pp. 187–200.

[19] KIM, M., COX, L. P., AND NOBLE, B. D. Safety, visibility, and performance in a wide-area file system. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (January 2002).

[20] KOZUCH, M., AND SATYANARAYANAN, M. Internet Suspend/Resume. In *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications* (Callicoon, NY, June 2002).

[21] LI, J., KROHN, M., MAZIRES, D., AND SHASHA, D. Secure untrusted data repository (SUNDR). In *Proc. of the 6th Symp. on Op. Syst. Des. and Imp.* (San Francisco, CA, December 2004), pp. 121–136.

[22] MICROSOFT CORPORATION. *Universal Plug and Play Forum*, June 1999. http://www.upnp.org.

[23] OSMAN, S., SUBHRAVETI, D., SU, G., AND NIEH, J. The design and implementation of Zap: A system for migrating computing environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (December 2002), pp. 361–376.

[24] RODRIGUES, R., CASTRO, M., AND LISKOV, B. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP)* (Banff, Canada, October 2001), pp. 15–28.

[25] SAPUNTZAKIS, C. P., CHANDRA, R., PFAFF, B., CHOW, J., LAM, M. S., AND ROSENBLUM, M. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating System Design and Implementation* (Boston, MA, December 2002), pp. 377–390.

[26] TOLIA, N., HARKES, J., KOZUCH, M., AND SATYANARAYANAN, M. Integrating portable and distributed storage. In *Proceedings of the 3rd Annual USENIX Conference on File and Storage Technologies* (San Francisco, CA, March/April 2004).

[27] TOLIA, N., KOZUCH, M., SATYANARAYANAN, M., KARP, B., BRESSOUD, T., AND PERRIG, A. Opportunistic use of content addressable storage for distributed file systems. In *Proceedings of the 2003 USENIX Annual Technical Conference* (May 2003), pp. 127–140.

[28] WALDSPURGER, C. A. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 181–194.

[29] WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. Scale and performance in the Denali isolation kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 195–209.