A Lightweight Secure Cyber Foraging Infrastructure for Resource-Constrained Devices

Sachin Goyal and John Carter
School of Computing
University of Utah
{*sgoyal, retrac*}*@cs.utah.edu*

Abstract

Resource-constrained embedded and mobile devices are becoming increasingly common. Cyber foraging, which allows such devices to offload computation to less resource-constrained surrogate machines, enables new and interesting applications for these devices. In this paper we describe a surrogate infrastructure based on virtual machine technology that allows resource-constrainted devices to utilize a surrogate's compute, network, and storage resources. After describing the design of our surrogate infrastructure, we demonstrate how it can be used to support real-time speech recognition and a synthetic web services application. Using a surrogate reduces the response time of speech recognition by a factor of 200 while reducing the energy drain on the client device by a factor of 60. Using a surrogate reduces the response time and energy drain on the client by factors of 21 and 25, respectively, for the web services application.

1. Introduction

In recent years, there has been an explosion of small computing devices, e.g., PDAs, cell phones, sensors, and a plethora of embedded devices. At the same time, network connectivity has become ubiquitous, even for these small devices. Such devices are typically resource-constrained, with limited energy, computation, memory, storage, and/or network resources. However, these limitations can be masked by utilizing the resources of less resource-constrained computers, a concept called *surrogate computing* or *cyber foraging* [15]. In a cyber foraging system, a surrogate machine makes its resources available to client devices to perform tasks on their behalf. We propose to use cyber foraging to let resource-constrained devices run applications and services that cannot be run on the small devices themselves due to lack of some resource(s). This paper describes a surrogate infrastructure that we are building to support this kind of cyber foraging.

Cyber foraging enables resource-constrained devices to "run" interesting resource-intensive applications that are beyond their own capabilities. For example, suppose you wanted to use speech or gesture recognition as inputs to a future PDA. Unfortunately, speech and gesture recognition are compute- and power-intensive operations, running far slower than real time on low-power devices [11]. Using cyber foraging, a PDA could capture the voice or gesture data and send it to a powerful surrogate computer to do the recognition. Alternatively, consider a future smart home environment, where tiny special-purpose embedded devices can utilize the resources of powerful desktop computers to perform interesting tasks that the devices themselves are incapable of performing. In addition to enabling new applications, cyber foraging can decrease the storage, battery, and computation requirements of embedded or mobile devices, thereby decreasing their size, complexity, and cost. Such devices would need only enough local resources to perform their common tasks, and could use surrogate resources to perform more complex or less common tasks.

Utilizing the idea of cyber foraging requires several challenges to be overcome, including:

1. developing mechanisms whereby a potential surrogate can make some of its resources available to resource-constrained clients,
2. providing a means for surrogates to advertise their availability and clients to locate surrogates with appropriate available resources,
3. developing a mechanism whereby clients can transfer tasks to the surrogate,
4. making the remote execution of surrogate tasks be (largely) transparent and easy to program, and

5. developing security and trust mechanisms so that surrogates can be assured that they (and their neighbors) will not be abused by surrogate computations.

We envision cyber foraging being used as follows. A PDA user clicks on the icon associated with an application configured to run, at least partially, on a surrogate machine. If the OS has not already obtained access to a surrogate, it invokes a surrogate acquisition module, which locates an appropriate surrogate, establishes a service contract with the surrogate for a particular amount of resources, and establishes a security context so that only the client can access the surrogate. In our system, the client is given `root` access to a virtual machine instance, and can download, install, and execute arbitrary applications or services, subject to negotiated resource limits. To invoke an application on the surrogate, the client ships a small program to a daemon listening at a known port on the surrogate, which runs the program on behalf of the client. Typically, this program is a shell script that downloads the real application over the Internet, installs it, and runs it. Once the surrogate portion of the application is installed on the surrogate, the application launches the client interface on the device (if any), transparently ships input data to the surrogate portion of the application, collects responses, and outputs them through the client user interface. We provide more details on the execution environment in Section 3.

A major goal of our work is to support cyber foraging on a conventional platform, without a large middleware layer, and demonstrate its use on real applications and systems. To do so, we build our surrogate framework on top of widely available technology. We use machine virtualization technology (VServer [1] and Xen [4]) to provide basic isolation. We use publicly available encryption technologies, both public and private key, as the foundation of our security and authentication infrastructure. We use a lightweight discovery protocol to locate potential surrogates.

To demonstrate the value of our cyber foraging framework, we evaluate it using a Sharp Zaurus PDA as a client and a RedHat Linux PC as a surrogate. We use cyber foraging to enable the Zaurus to perform continuous speech recognition using the sphinx2 [8] system from CMU and to perform a synthetic web services application wherein the surrogate sifts through 19 megabytes of data on behalf of the client. We find that using a surrogate reduces the response time of the compute-intensive speech recognizer by a factor of 200, while reducing the energy drain on the Zaurus by a factor of 60. For the network-intensive synthetic web services, using a surrogate reduces the response time by a factor of 21, while reducing the energy drain on the Zaurus by a factor of 25.

Our main contributions are as follows:

- We build a practical system to support cyber foraging with realistic notions of trust, security, and usability.
- We demonstrate the value of using virtual machine technology as the basis for building a surrogate infrastructure.
- We show that interesting applications can be enabled even using only trusted computers available to us (e.g., our home PC or nearby office PC).
- We demonstrate experimentally the performance and energy saving potential of cyber foraging for a simple system involving a Sharp Zaurus client and a Linux PC surrogate.

2. Related Work

The basic idea of using surrogates to support pervasive computing was introduced in Satyanarayanan's paper on the challenges of pervasive computing [15]. To support this vision, Flinn *et.al.* built Spectra [5, 6], a remote execution environment that allows a mobile device to use the processing power of a nearby surrogate computer. Spectra runs on top of Coda [9] and Odyssey [13]. Spectra monitors application resource usage and the availability of resources in the local environment to decide when and where to utilize cyber foraging. Balan *et.al.* [3] extended Spectra to support application partitioning, and show that application-specific knowledge regarding how to partition an application between a client and a surrogate can be captured in a compact form.

Our work is orthogonal to these efforts. Our focus is on building a cyber foraging infrastructure out of commodity systems and establishing secure surrogate sessions dynamically. We do not require clients and surrogates to share a file system (e.g., Coda) or require the client to run a relatively heavyweight middleware system. Our clients and surrogates do not share a common file system, so we rely on the surrogate being connected to the Internet to locate and download client application code. We also address security issues to allow only authorized users to utilize the services of the surrogates.

Messer [12] developed a system for dynamically partitioning Java programs and offloading pieces of them to surrogates based on memory and processing constraints. However, their approach entails significant networking overhead, and thus energy consumption, because the client and surrogate portions of program are highly coupled.

In contrast to prior work on automatic application partitioning, we rely on application writers to partition applications at development time and assume that applications are split in a way that mitigates client-surrogate communication. Thin clients have limited resources and IO interfaces, so developers writing applications for these devices already tune them carefully. Using our simple surrogate-enabling

library, developers can easily build two versions of their applications, a low-fidelity option for when no surrogate is available and a high-fidelity one that exploits surrogates. We expect that a common strategy will entail running most of the application on the surrogate and using the client for input-output. Cyber foraging is most useful for applications that can be divided into loosely-coupled components; otherwise the cost of communication between the client and the surrogate will offset the performance or energy benefits of using a surrogate. Conveniently, many applications fall into this category, including perception processing (e.g., speech and gesture recognition), data mining (e.g., semantic web and web services operations), sensor aggregation, distillation proxies, and many smart home applications.

3. System Design

Our goal is to create a system that can be deployed in existing computing environments to enable cyber foraging without requiring a heavyweight middleware system.

3.1. Overview

In our system, there are *clients* and *surrogates*. Clients are typically resource-constrained devices with some form of networking capability, e.g., a PDA, mote, or cell phone. Potentially constrained resources include energy, compute power, storage, and network bandwidth. Note that while the examples in this paper assume clients with limited capabilities, a client could be a desktop computer that wishes to harness the compute capabilities of hundreds of other desktop computers, ala the "Grid." Surrogates are typically resource-rich devices that are willing to run programs and/or provide storage on behalf of clients. Surrogates might be a desktop PC available via the local wireless LAN (e.g., in a smart home, office, or commercial hotspot environment) or might be your home PC or a commercial surrogate connected via the Internet. For our work, we assume that the surrogate is connected to the Internet via a high-speed network. Figure 1 shows a simple cyber foraging scenario where a PDA client moves some subtask to a local surrogate machine.

For this design to be useful and gain wide acceptance, many challenges must be overcome, including:

1. Clients should be able to specify their requirements for a surrogate server easily and locate a suitable surrogate in real time.
2. Surrogates must be able to limit the use of their resources (e.g., cpu time, memory, file storage, and network bandwidth) by client applications.
3. Clients should be able to install and execute arbitrary applications and services.

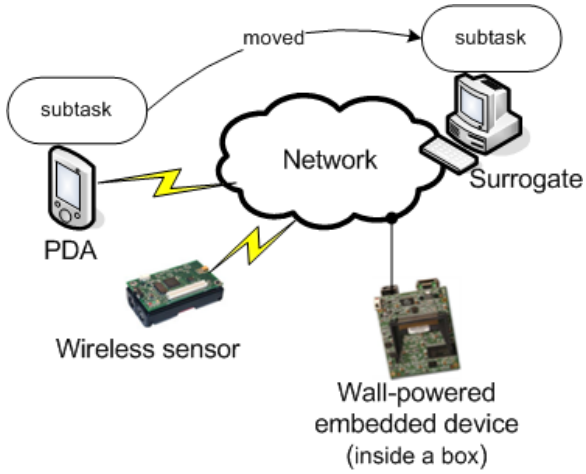

Figure 1. A cyber foraging scenario

4. The application interfaces for cyber foraging should be simple and easy to use.
5. There must be sufficient security and privacy guarantees that both clients and surrogates will be willing to take part in the infrastructure.

We discuss our solutions to these challenges throughout the remainder of this section. We are investigating solutions to other challenges, such as developing a trust model that allows entities to decide how to operate based on how much they trust each other and developing a cost model to support resource management and enable commercial surrogates, but this work is beyond the scope of this paper.

3.2. Surrogate Design

Surrogates can be implemented in multiple ways. We choose to use *virtual machine* technology. Virtual machine technology allows a single surrogate machine to run a configurable number of independent *virtual servers* with greater greater *isolation*, *flexibility*, *resource control*, and *ease of cleanup* than simply running surrogate processes directly on top of a standard Unix box. In terms of isolation, client applications running on different virtual machines cannot directly interfere with one another, nor can they access resources reserved for the surrogate machine. Different clients can install arbitrary applications, packages, and even kernel patches, without interfering with other clients or the base surrogate. Letting clients install arbitrary code on their virtual machine provides tremendous flexibility – what they can do is not limited by what some middleware layer supports. The virtual machine monitor can enforce resource controls (e.g., disk quota, cpu share, and physical memory

allocation) on a per-VM basis, which allows allows individual clients to share the surrogate machine fairly. This design allows normal work to proceed on the surrogate without undue impact by surrogate clients. Finally, it is easy to clean up after a client – the surrogate host system simply shuts down the virtual server instance and restores the associated disk partition to its original (clean) state.

Currently we support two types of virtual machines: *Linux-Vserver* [1] and *Xen* [4]. Virtual servers under Linux-Vserver share a single kernel instance. Isolation is provided by OS modifications that encapsulate groups of processes called contexts. Each virtual server runs in its own root file system (`chroot`'ed), is provided its own IP address, and can only access the files, processes, and IPC primitives owned by it. Xen para-virtualizes an x86 platform, which allows multiple independent OS kernels (currently Linux or BSD[1]) to run on a single machine. Each OS instance accesses hardware through virtualized device drivers.

In general, Xen provides a higher degree of isolation and security than Vserver, at the cost of additional run time overhead and a slower startup. The Xen VM monitor allows us to allocate physical memory between various virtual servers and supports CPU schedulers that fairly share the CPU cycles. Vserver currently provides only basic hard limit controls for CPU and memory sharing, but efforts are underway to add functionality developed as part of the class-based kernel resource management (CKRM) project [2]. We plan to use Xen when clients and surrogates do not fully trust one another or we require more comprehensive resource controls and to use Vserver for mutually trusting clients and surrogates.

Language based virtual machines, e.g. Java, can provide some of the same features, but restricts the flexibility of the system to use programs written in that language only.

3.3. Service Discovery

To utilize the surrogate infrastructure, clients must first locate a suitable surrogate with the help of a service discovery system. We employ a simple service discovery server that allows surrogates to register themselves using an XML-like description of their capabilities, e.g.,

```
<service> surrogate
    <OS> Linux
         <distribution> Redhat
             <version> 9.0 </version>
         </distribution>
         <distribution> Debian
             <version> 3.0 </version>
         </distribution>
    </OS>
</service>
```

[1] A Windows XP version exists, but is not publicly available.

In this example, the surrogate offers clients instances of two Linux distributions, Redhat 9.0 and Debian 3.0. Clients locate surrogates by querying the service discovery server using a similar XML-like notation. The service discovery server matchs requests against registered surrogates. In the future, we plan to adopt an existing service discovery mechanism, e.g., the Service Location Protocol [7] and extend our lookup mechanism to consider issues such as surrogate load.

Currently we expect programmers to make conservative estimates of application resource requirements based on application knowledge or simple profiling. For example, for the speech recognition application a coarse estimate could be obtained by running `top` and observing typical CPU and memory usage patterns, and then a modest margin of error (e.g., 25-50%) could be added to the request. In the future, we plan to investigate the value of more sophisticated profiling and history-based tools. Accurately predicting the resource requirements of an application is an area of active ongoing research area in both the embedded and computational grid domain that we plan to follow closely.

3.4. Control Flow

Figure 2 shows a typical client-surrogate control flow:

1. **Service Discovery Request**: The client sends a message to the service discovery server to find an appropriate surrogate.

2. **Service Discovery Reply**: The service discovery server responds with the IP Address and port number of the surrogate's *surrogate manager process*, which listens for client connection requests.

3. **Service Start Request**: the client connects to the surrogate manager process and requests a virtual server with specific resource guarantees. The surrogate manager first authenticates the client (see Section 4). If the client is authorized to use this surrogate, the surrogate manager processes the client resource request (described via an XML-like notation) and, if adequate resources are available, establishes a contract to provide service for a fixed duration. The maximum number of virtual server instances and the maximum machine resources that a surrogate is willing to provide to clients are configurable.

4. **Virtual Server Start**: The surrogate manager maintains root partition images for each available OS (e.g., Linux Redhat 9 or BSD). In response to an authorized client request, it dynamically initializes a preallocated root partition with the appropriate root image and starts a new virtual server. It takes more than a minute to initialize a one GB root partition. To reduce this setup

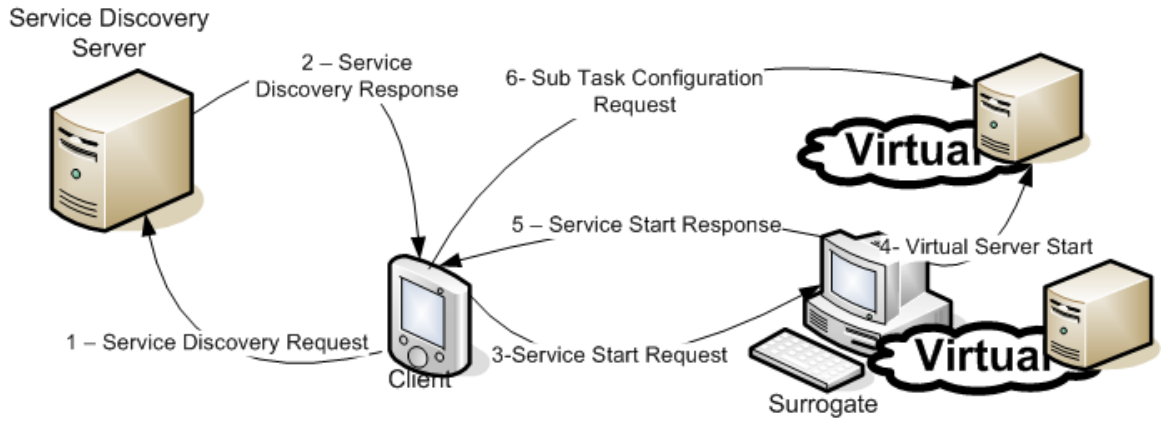

Figure 2. A typical client-surrogate control flow

overhead, we maintain a pool of preinitialized partitions to hide the root image copy time in the common case. Each virtual server is allocated its own IP address.

5. **Service Start Response**: Once the new virtual server is running, the surrogate manager returns the IP address of the virtual server to client.

6. **Sub Task Configuration Request**: Each virtual server is configured to run a *virtual server manager* (VSM) that handles requests for surrogate operations from its client. To invoke an operation on a surrogate, the client sends a Sub Task Configuration Request to the virtual server manager. Client requests consist of a URL that points to a program that the client wants the VSM to run on its behalf. Typically, the program is a shell script that downloads the necessary software/packages, installs them, and then invokes them. For example, in Section 5 we describe an experiment that installing a Sphinx2 voice recognition server on the surrogate and having it accept client voice recognition requests on a well known port. In addition the VSM, the virtual server runs a secure shell daemon (`sshd`) set up so that the client can log on as `root` and manually install and invoke any operations they wish.

7. **Service Termination**: When the client explicitly terminates the surrogate or when the reserved interval expires, we clean up the virtual server by reinstalling its root file system with a clean image. Wiping the partition removes any changes made by the surrogate, thereby preserving privacy and protecting future clients from any malicious software installed by the client.

We anticipate that some clients will use the same surrogate to perform the same operation multiple times, e.g., a mobile user may use their home PC as a surrogate when traveling. To improve the performance of this common scenario, we let trusted users save customized root partition images with all necessary packages, applications, and system boot up scripts on the surrogate. We allow the same user (identified by their public key) to use request this image be installed when they request a virtual server. We are planning to add a surrogate cache whereby surrogates cache packages that are often downloaded so that they can be retrieved locally. Both of these optimizations reduce the startup overhead of instantiating a surrogate virtual server.

3.5. Client Interface

To support the functionality described above, we provide a simple client library that supports the following operations:

- **get_surrogate()**: Takes as input the IP address of a surrogate server and a service description string. Returns a success code. If successful, the IP address of the virtual server is also returned. If the client has an active surrogate server, its IP address is returned. Otherwise, a new virtual server is allocated as described in Section 3.4 and its IP address is returned.

- **subtask_conf_request**: Takes as input the URL where a program that the client wishes to run can be found, which is sent as a subtask configuration request to appropriate virtual server manager. Returns a success code.

- **get_virtual_server_ip**: Returns the IP address of the active surrogate virtual server, if any.

To enable client-side scripting, these functions are also provided as programs and the configuration information (e.g., IP address of current surrogate server) is available via environment variables. In addition, as described in Section 3.4, users can use `ssh` to login to the active virtual server as `root` and invoke operations directly.

If the client wishes, a single virtual server can support multiple clients or multiple users. The multiple clients can share the same private key (e.g., using the certificate mechanism described in Section 4), or the first client can install addition public keys in the database of authorized clients for this virtual server. Similarly, multiple users can be supported by having the first user, who has `root` access, create additional accounts for additional users. This capability lets clients instantiate servers on the surrogate that can be accessed by multiple users, e.g., a game server that other people can contact and use.

4. Security

Two basic problems that must be addressed in a surrogate infrastructure are (i) how to ensure that only authorized clients can allocate a virtual server and (ii) how to ensure that a surrogate virtual server is usable only by the client for which it was created. Currently we assume a pre-existing trust relation between clients and surrogates, e.g. a user using his home/office computers as surrogates or a university/company providing surrogate services to its employees. Commercial deployment of surrogate services, e.g., allowing a roaming user to acquire surrogate services from on an untrusted for-pay surrogate situated at a nearby cyber cafe, would require features like trust models and payment systems that we do not currently address. We plan to explore such mechanisms as future work. We envision a wide diversity of devices with different operating systems using our infrastructure, so we employ widely available and verified cryptography and authentication solutions whenever possible

Instead of selecting any specific authentication mechanism, we provide a simple but flexible authentication framework that can support multiple mechanisms. Clients can specify their preferred authentication mechanism (`authtype`) as part of the Service Start Request sent to the surrogate manager. Currently we support only a single authentication mechanism, but plan to add support for additional mechanisms as future work. For example, we are considering adding support for authentication based on shared secret keys to support devices for which public key encryption is too compute-intensive, e.g., motes. In this section, we discuss the authentication mechanism that we employ in the current prototype.

4.1. Public Key based Authorization

Our current prototype supports a simple authentication mechanism whereby surrogates maintain an authorization list containing the public keys of authorized clients. We chose to employ a public key base system because it is easy to distribute public keys without worrying about confidentiality. Our current protocol utilizes SSL/TLS and SSH for communication with the surrogate manager and the virtual server respectively, because they are well understood and widely available secure protocols. Also, we envision many end-users using `ssh` for interactive surrogate sessions, because it allows them to manually configure the virtual servers via a well understood shell interface.

To initiate a surrogate request, the client establishes a SSL/TLS session with the surrogate manager. We employ a low overhead TLS cipher (TLS_RSA_WITH_NULL_MD5) and and client side authentication to provide endpoint authentication and to ensure the integrity of data exchanged [14]. We do not encrypt the data sent in this session, because confidentiality is not necessary and bulk encryption would impose a significant performance and energy load on resource constrained clients.

Once the session is established, the surrogate manager checks to see if the client's public key is in its list of authorized clients. If it is, the surrogate manager allocates a new virtual server for the client. Before starting the virtual server, the surrogate manager adds the client's public key to the `/root/.ssh/authorized_keys` file in the virtual server's root directory. This public key is used by the virtual server to determine which clients are authorized to use it. In addition, by placing the client's public key in this file, the client can login to the server directly using a normal `ssh` client, without a password, and manually install and start services using normal command line tools. The client uses the same RSA key pair for TLS sessions with the surrogate manager and subsequent `ssh` sessions with the virtual server, because it allows the client to manage only one key pair instead of two. For TLS sessions, the client can use a self-signed certificate without compromising security because the surrogate manager ensures that the client's public key is in its authorized list.

After allocating the new virtual server instance, the surrogate manager sends the IP address of the virtual server along with its public host key to the client, which stores the public key in its `known_hosts` file. Transferring the public host key of the virtual server to the client ensures that subsequent `ssh` sessions established from the client to the virtual server are secure. Currently clients do not authenticate servers. For the surrogate server to authenticate itself to the client, the server must have its certificate signed by a known authority, e.g., a certificate authority, which we do not currently support. In the future we plan to have the ser-

vice discovery server provide the public key of the surrogate along with its IP address to eliminate the potential problem of surrogate masquerading. In such a case, the surrogate server can also use a self-signed certificate.

To perform a Sub Task Configuration Request, the client uses `ssh` to remotely execute the virtual server manager (VSM) on the surrogate virtual server. The arguments to the VSM are a URL and the MD5 checksum of the file specified by the URL. In response, the VSM downloads the program from the URL and verifies the MD5 checksum before running it. In practice, the program usually consists of a shell script that downloads the necessary software/packages, installs them and then invokes them.

4.2. User Certified Client Devices

The simple public key authorization mechanism described above is effective, but it requires that each surrogate server be configured with a list of all authorized clients' public keys. As the number of servers and client devices grows, maintaining these lists will become a nuisance or could drive people to reduce security by using the same keys on all devices.

To address this problem, we support user certified client devices wherein each user has a single global public/private key pair that is used to identify them to potential surrogates. Surrogate servers maintain a list of the public keys of authorized *users*, not authorized client devices. When a user installs a new device, the user signs the new device's certificate. Subsequently when the device contacts the surrogate manager, it uses the user-signed certificate instead of a self-signed certificate to prove that the device belongs to the user. As a result, the surrogate server need only be configured with a list of the public keys of authorized users, not a separate key for each authorized client device.

5. Experimental Evaluation

In this section we evaluate our cyber foraging infrastructure using two applications, the `sphinx2` speech recognition system and a synthetic web services application. We chose these two applications to investigate the value of offloading work to surrogates for both compute-intensive applications (`sphinx2`) and network-intensive applications (the synthetic web services application).

5.1. Experimental Setup

For our experiments, our surrogate platform is a Dell Dimension 4550 Series Computer with a 2.40-GHz P4 processor and 512MB of RAM. The client is a Sharp Zaurus SL-5500 PDA running Linux 2.4.6-rmk1-np2-embedix (OpenZaurus distribution 3.2). The Zaurus has a 206-MHz SA1110 processor, 16MB of FlashRAM, and 64MB of DRAM, 24MB of which is dedicated to file storage. It connects to its LAN via a Linksys WCF12 compact flash 802.11b wireless card. For all experiments, the surrogate machine is connected directly to the CS department LAN. To test the importance of physical proximity between the client and surrogate, we move the client between the CS department LAN and the first author's home LAN, which is connected to the Internet via a cable modem. When the Zaurus is connected to the CS department LAN, the RTT between the client and surrogate varies from 2-3ms, whereas when the Zaurus is connected to the home LAN, the RTT between the client and surrogate varies from 72-73ms.

For each of our two applications, we run two sets of experiments, one in which the application is run in its entirety on the PDA and one in which the resource-intensive portion of the application is dynamically instantiated on the surrogate node. We first report the time required to initialize the surrogate service, including the time to perform service discovery, instantiate a new virtual server, and install and start the test application. We then report the run times and energy consumed by the PDA to run the application both locally and with the help of a surrogate. For experiments where we run the application entirely on the PDA, we disable the network card and LCD backlight to minimize energy drain. In contrast, when we execute the resource-intensive portions of the application on the surrogate, we leave the network card enabled, which results in a conservative estimate of the energy requirement. In both situations, the PDA is idle other than the test application and an instance of `top`, a system management tool that we use to extract CPU and memory utilization.

To determine the amount of energy consumed by each experiment by the PDA, we use its power management interfaces to extract coarse-grained battery measurements (e.g., start at 100% battery life and end at 85% battery life). Because the battery appears to discharge in a non-linear fashion, we fully charge the battery before each experiment and run the experiment enough times to drain at least 15% of total battery capacity. We then report the average amount of energy consumed by a single run.

5.2. Sphinx Speech Recognition

`Sphinx2` [8] is a real-time, large-vocabulary, speaker-independent speech recognition system developed at CMU. It uses pre-made acoustic models for American English: an acoustic model, a pronunciation dictionary, and a language model. These models are stored in several files that in aggregate are roughly 23MB in size. When `sphinx2` is loaded and run completely on the Zaurus, storing 23MB worth of model files requires deleting most other applications and user files from the Zaurus's tiny file server. In

contrast, the client stub required to run `sphinx2` on the surrogate server is only 12KB in size.

To perform this experiment, we created two versions of `sphinx2`, one that ran entirely on the Zaurus and another that split the functionality between the Zaurus and the surrogate. For both, we used the `sphinx-2.0.4` source code from sourceforge. Porting `sphinx2` to run on the Zaurus required non-trivial effort because it uses non-standard sound driver settings and the Zaurus sound driver did not support many of the `ioctl` calls used by `sphinx2`. Once ported, it was relatively straightforward to divide the application into two components for use in our surrogate infrastructure. The client portion uses the system calls described in Section 3.5 to allocate a virtual server and instantiate and instance of the `sphinx2` server. It then records what the user says and sends the raw digitized sound data to the surrogate for analysis. We believe this development experience will be typical for surrogates – most of the work is involved in porting your application to the small device, whereas splitting the application to exploit cyber foraging is straightforward.

Client Location	Linux-Vserver	Xen
Univ	4.22 (0.44)s	12.43 (1.78)s
Home	4.41 (0.35)s	12.57 (1.57)s

Table 1. Average response time for allocating and initializing a virtual server. Standard deviations are in parentheses.

Client Location	Linux-Vserver	Xen
Univ	.37 (.043)s	.30 (.021)s
Home	.78 (.027)s	.74 (.025)s

Table 2. Average response time for instantiating `sphinx2` speech recognition engine. Standard deviations are in parentheses.

5.2.1. Experiments For our first set of experiments, we measure how long it takes to acquire a new virtual server and how long it takes to load and instantiate the `sphinx2` surrogate on the newly allocated virtual server. We use public key authentication for these experiments. Prior to the experiment, we enter the user's public key in the surrogate machine's list of allowed clients. For all experiments, the input utterance was a pre-recorded sound sample of the phrase, "'*Go Forward 10 meters*".

Table 1 shows the time needed to acquire a virtual server for both flavors of surrogate (Vserver and Xen). The time to allocate and initialize a new virtual server is largely independent of whether the client is co-located on the same LAN or connecting remotely, because most of the time is due to virtual server startup. It takes about three times as long to allocate and initialize a virtual server on Xen compared to Vserver (12.4-12.6 secs versus 4.2-4.4 secs), because Xen needs to boot a new kernel whereas virtual servers on Vserver share a single kernel. The virtual server acquisition is done only once during a session, so this 4-12 second overhead is largely transparent to the user. The startup overhead could be reduced by keeping a pool of pre-booted virtual servers on the surrogate, ready for allocation.

Table 2 shows the time to download and instantiate a `sphinx2` surrogate, which entails downloading the `sphinx2` server software (6.3MB) and starting the server. For this experiment, the server software was stored on on the same LAN as the surrogate computer. It takes well under a second in all circumstances - the slightly higher overhead of instantiating `sphinx2` when the PDA is remote is due to the higher latency between the client and surrogate.

Table 3 compares the results of running the speech recognizer on the Zaurus and on the surrogate machine. The performance differences are striking. When run on the Zaurus, it takes almost two minutes to recognize the 4-word utterance, far slower than real time. In contrast, the surrogate version can recognize the phrase in under a second when the client and surrogate share the same LAN, or in just over two seconds when invoked remotely. The slower response time when the operation is invoked from a home LAN is due almost entirely to the low upload bandwidth cap imposed by the cable modem server (roughly 25 KB/sec). The raw audio data file is 44 KB in size and requires just under two seconds to transfer to the remote surrogate, which closely matches the difference in response times.

In addition to dramatically reducing the recognition time, using a surrogate also dramatically reduces resource consumption on the Zaurus. As mentioned above, simply storing the `sphinx2` application and model files on the Zaurus required deleting most of the other applications and data from the Zaurus. Also, when run locally, `sphinx2` utilized more than 95% of the Zaurus' CPU throughout the experiment, spiking as high as 98.8%, and occupied more than 50% of its memory. This left the PDA unusable for any other activity, and resulted in a popup warning, "*The Memory is very low, Please end this application immediately!*". In contrast, using a surrogate reduced the CPU and memory overheads to 0.3-0.5% and 1.1%, respectively, leaving the Zaurus free to perform other tasks.

When we compare the energy cost of invoking the speech recognizer locally versus on the surrogate, the results again strongly favor using a surrogate. Performing speech recognition at a local surrogate consumes roughly 60 times less PDA energy while performing speech recognition

Type	Client - Server	Response Time	CPU Util	Memory Util	App Size	Battery Util
local	-	117.49 (0.96)s	>95%	51.6-55.9%	23MB	1.1 (0.08)%
cyber foraged	Univ-Xen	0.69 (0.017)s	0.3-0.5%	1.1%	12KB	.018 (0.001)%
	Univ-Vserver	0.59 (0.021)s				
	Home-Xen	2.24 (0.018)s				.083 (0.006)%
	Home-Vserver	2.31 (0.024)s				

Table 3. Sphinx2 speech recognition on the Zaurus. Standard deviations are in parentheses.

at a remote surrogate requires roughly 13 times less PDA energy. These results come despite the fact that transferring data across the wireless link requires a non-trivial amount of energy. In addition, we do not use any power saving mode for wireless card during the surrogate experiments, which introduces energy drain not directly related to the experiment. On the flip side, the PDA consumes roughly 0.3% of its battery power when idle for two minutes, so a portion of the 1.1% consumed by the local speech recognition experiment constitutes a fixed cost. Nevertheless, it is clear that for compute-intensive operations like speech recognition, cyber foraging has the potential to generate significant energy savings.

In summary, these results clearly show that using a surrogate can lead to dramatic improvements in both response time and energy consumption for compute- and storage-intensive applications like `sphinx2`. Cyber foraging should be even more effective at reducing client energy consumption if used in combination with a wireless LAN driver optimized for energy consumption, e.g., by using power saving modes to reduce energy consumption while the network is idle [10].

5.3. Data Mining

Data mining or web services, which entail sifting through potentially large amounts of data acquired from one or more storage servers, are also well suited to cyber foraging. In contrast with speech recognition, these types of applications tend to be bandwidth-intensive rather than compute-intensive. The amount of computation done per-byte of data is much lower than for speech recognition. Thus, the potential benefits of using a surrogate to perform the data filtering include both the reduced processing time and also reducing the amount of data transferred over the energy-hungry wireless LAN.

To evaluate the impact of using a surrogate on this class of applications, we designed the following synthetic benchmark. The benchmark entails downloading three 6.3 MB files, performing an MD5 message digest operation on each file, and outputting the resulting three checksum values. For this experiment, the Zaurus and surrogate run on the same LAN. When a surrogate is used, the client sends it URLs for the three files, the surrogate downloads the three files and computes their MD5 checksums, and then the surrogate returns the resulting values to the client. When the surrogate is not used, we disable the wireless LAN card after the files have been downloaded to conserve energy, and then perform the MD5 calculations locally.

Type	Response Time	Battery
local	61.47 (1.29)s	1.5 (0.109)%
cyber foraged	2.9 (0.066)s	0.06 (0.003)%
cyber foraged (20s sleep)	24.4 (0.648)s	0.1 (0.008)%

Table 4. Synthetic Benchmark Results. Standard deviations are in parentheses.

Table 4 presents the results of these experiments. In the default configuration, the web site hosting the three files resides on the same LAN. In this case, downloading the files to the Zaurus and performing the MD5 checksums locally takes 21 times as long and consumes 25 times more battery energy. We then modified the surrogate experiment to model the situation where the files are located across the Internet and a non-trivial amount of time is required to download them. For this experiment, the client disables its wireless LAN card and sleeps for 20 seconds after making the request to the surrogate to conserve energy. After it wakes up, it queries the surrogate for the results of the message digest operations. In this case, the client consumes 15 times less energy than if it had performed the checksum locally, and half of that energy is the baseline energy drain for 24 seconds of operation. Again, despite this application having a very different computation to communication ratio, cyber foraging proves to be very valuable in terms of reducing response time and client energy consumption.

6. Conclusions and Future Work

In this paper, we have described the design and implementation of a lightweight secure cyber foraging infrastructure based on virtual machine technology. Our system

allows us to forage compute, memory, storage, and network resources securely from a surrogate PC. The surrogate mechanisms we propose are lightweight, do not require clients to run any special middleware software, and generic. The preliminary experimental analysis show that cyber foraging has great potential for reducing the response time and energy requirements of running complex applications on (or for) resource-constrained devices. In particular, cyber foraging enabled a PDA to perform real-time speech recognition, which was impossible using only the PDA's resources. The growing interest in virtual machine technology will likely reduce the performance overheads and could lead to future Linux distributions where surrogate support (in the form of a simple daemon) is standard.

We are working on ways to extend our work to untrusted and wide area environments. For example, commercial wifi hotspots could be augmented to support surrogate computing, enabling users with small devices like cell phones access to significant compute and storage resources. Perhaps more interesting, wide area deployment of a cyber foraging infrastructure could give clients access to either a multitude of servers on which they could perform Grid-like computations, or let clients invoke significant computation near large data repositories like the SkyServer or TerraServer.

Using virtual servers and flexible authentication policies addresses some, but not all, of the issues that arise in an untrusted or wide area environment. Current virtual machine monitors do not provide administrators with sufficient control over their surrogate resources, especially in terms of the network – without adequate controls, running a surrogate on your site could let attackers circumvent your firewall protection or let them set up spam zombies. To support pervasive (and especially commercial) deployment of surrogates, we need to develop a cost model that enables clients and servers to negotiate a "cost" for allocating a virtual server, e.g., as part of a peering agreement or a micropayment. In addition, the authentication and cost models will need to incorporate the notion of varying degrees of trust, ranging from fully trusted (a user's home or office PC) to semi-trusted (a surrogate provided by a company with which the user has a business relationship) to untrusted (a random user or surrogate). Ultimately, we envision a system where, in addition to providing clients access to a user's personal surrogates, clients can negotiate very specific resource and location requirements (e.g., "I need access to a 200 MFLOP/s of computing on a machine with sub-20msec latency and at least 30Mb/s throughput connectivity of Google.com, and am willing to pay up to $0.03 per minute for this service.")

Although many challenges must be overcome before our long term vision will come to fruition, our work has demonstrated the basic value of cyber foraging and shown how it can be used to enable interesting applications on small (potentially embedded) devices. Even restricting surrogates to the small set of fully trusted computers available to a user or device, e.g., their home and office PC, enables interesting applications and services without any new investment in hardware or new security risks.

References

[1] Linux vserver project, available at http://www.linux-vserver.org/.

[2] Class-based kernel resource management (ckrm), http://ckrm.sourceforge.net/.

[3] R. K. Balan, M. Satyanarayanan, S. Park, and T. Okoshi. Tactics-based remote execution for mobile computing. In *1st Intl. Conf. on Mobile Systems, Applications, and Services*, 2003.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. of the 19th ACM Symposium on Operating Systems Principles*, October, 2003.

[5] J. Flinn, D. Narayanan, and M. Satyanarayanan. Self-tuned remote execution for pervasive computing. In *HotOS-VIII*, pages 61–66, 2001.

[6] J. Flinn, S. Park, and M. Satyanarayanan. Balancing performance, energy, and quality in pervasive computing. In *Proc. of the 22nd Intl. Conf. on Distributed Computing Systems*, July, 2002.

[7] E. Guttman. Service location protocol: Automatic discovery of IP network services. *IEEE Internet Computing*, 3(4):71–80, 1999.

[8] X. Huang, F. Alleva, H.-W. Hon, M.-Y. Hwang, K.-F. Lee, and R. Rosenfeld. The SPHINX-II speech recognition system: an overview. *Computer Speech and Language*, 7(2):137–148, 1993.

[9] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. In *13th ACM Symposium on Operating Systems Principles*, 1991.

[10] R. Krashinsky and H. Balakrishnan. Minimizing energy for wireless web access with bounded slowdown. In *The 8th Intl. Conf. on Mobile Computing and Networking*, 2002.

[11] B. Mathew, A. Davis, and Z. Fang. A Low-Power Accelerator for the SPHINX 3 Speech Recognition System. In *Proc. of the Intl. Conf. on Compilers, Architecture and Synthesis for Embedded Systems*, Oct 2003.

[12] A. Messer, I. Greenberg, P. Bernadat, D. S. Milojicic, D. Chen, and T. J. Giuli. Towards a distributed platform for resource-constrained devices. In *Proc. of the 22nd Intl. Conf. on Distributed Computing Systems*, 2002.

[13] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker. Agile application-aware adaptation for mobility. In *Proc. of the 16th ACM Symposium on Operating System Principles*, Oct, 1997.

[14] E. Rescorla. *SSL and TLS - Designing and Building Secure Systems*. Addison-Wesley, 2001.

[15] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, August 2001.