Volume: 08 Issue: 06 Pages: 3274-3282 (2017) ISSN: 0975-0290

A Survey of Cyber foraging systems: Open Issues, Research Challenges

Manas Kumar Yogi
Department of Computer Science and Engineering, Pragati Engineering College(Autonomous)
Email: manas.yogi@gmail.com
Darapu Uma
Department of Computer Science and Engineering, Pragati Engineering College(Autonomous)
Email: umadarapu03@gmail.com

ABSTRACT

This paper presents a survey on current applications which practice the pervasive mechanism of cyber foraging. The applications include the LOCUSTS framework, Slingshot, Pupetter. This applications advocated the operating principle of task sharing among resource deficient mobile devices. These applications face some design issues for providing efficient performance like task distribution and task migration apart from the security aspect. The general operating mechanism of the cyber foraging technique are also discussed upon and the design options to leverage the throughput of the inherent mechanism is also represented in a suitable way.

Keywords - **Cyber foraging, resource deficient, LOCUSTS, Sligshot, Pupetter, Task migration.**

Date of Submission: May 01, 2017 Date of Acceptance: May 16, 2017

I. INTRODUCTION

Cyber foraging refers to pervasive computing mechanism where resource deficient, mobile devices offload some of their heavy task to stronger surrogate machines in the surroundings. The term cyber foraging was given by Mr. M. Satyanarayanan in his 2001 paper. Cyber foraging, helps the mobile devices to take on more resource intensive tasks by leveraging unused resources on larger computers in the vicinity. Cyber foraging is foraging for a multiple resource types , not limited to processing power. Among the resources that can be foraged for is network connectivity, storage, processing power, bandwidth and much more. All of these resource types are equally important in a cyber foraging scenario. There are many possible usage scenarios where cyber foraging can be utilized. Some designs for pervasive computing advocates for wearable computing devices which include small computing devices that may be worn by their users like clothes[1]. Users of such devices are not interested in carrying around equipment which are heavy, and these devices must therefore be as lightweight as possible. This is opposite to the user's need to have as powerful a device as possible. The desired computing power can be added to these small wearable devices through techniques such as cyber foraging. Consider the following situation : a doctor doing house calls is wearing a small headset (similar in size and form to the well-known Bluetooth headsets for mobile phones). Using this headset he would like to be able to enter data pertaining to his patients into an electronic journal. This means that the headset is faced with the difficult task of continuous voice recognition. The headset is unable to perform this translation task by itself, so instead of performing the actual voice recognition it only records the words uttered by the doctor. Whenever the headset comes within range of usable computing resources it forwards some of the recordings to these machines who respond by returning the translated text[2]. In case the surrogate has an Internet connection it may even be given the task of updating the patient's journal directly. After translation the headset may discard the recording and thus free storage for additional recordings. In the preceding scenario the application running on the mobile device works in two modes; high fidelity and low fidelity. When no surrogates are within range the headset simply saves the recordings (low fidelity), and when surrogates can be used the recordings are immediately translated into text (high fidelity). This high/low fidelity aspect is inherent in all cyber foraging applications, when surrogates are available high quality work may be done, but this does not mean that the applications will only work in the presence of surrogates. For cyber foraging to be usable a low fidelity setting must also be possible, where the mobile device itself is running the application, but at a diminished fidelity. In the scenario low fidelity means that the headset only stores the recordings, but it would also be possible to ask the mobile device to do the processing itself, or even to do it in combination with other mobile devices that reside in the doctors personal area network. To be able to perform the actions described above a number of things are needed. At first the mobile device must be able to monitor the network looking for any available surrogates[3]. Once found the mobile device must be able to distribute tasks to surrogate machines, and, in the case that the user is moving while tasks are being performed, surrogates must be able to migrate tasks between each other so that the result may be returned. This amount to complex operations, and it should not be the work of the application programmer to implement this.

Volume: 08 Issue: 06 Pages: 3274-3282 (2017) ISSN: 0975-0290

2. CURRENT RESEARCH CHALLENGES

There are a considerable number of challenges that must be addressed when designing a framework for cyber foraging.

2.1 Distribution of task

How can the complex tasks be delegated to surrogates, and exactly up to what complexity of tasks should be moved onto the surrogates.

2.1.2 Migration of task

When mobile devices are using surrogates the tasks that are distributed to surrogates must be transferrable. It must be possible to move running tasks between surrogates and also to move a task back to the mobile device. Apart from this remote execution specific challenges a number of other challenges are posed as well in a cyber foraging framework, challenges such as device discovery, capability announcement, data staging[4]. One final important challenge for cyber foraging is security. Surrogates must execute code on behalf of, possibly unknown and thus untrusted, mobile devices and data must be transmitted over wireless links that are easy for an eavesdropper to monitor. Finally, the client must be able to trust that the surrogate actually performs the task that it is asked to. How can this be done in a secure manner? A fine balance between security and flexibility must be found here.

2.2 Cyber Foraging Steps

A cyber foraging approach includes some steps that every available cyber foraging systems have considered all or some of them. These steps can be summarized as follows.

2.2.1 Surrogate discovery

At the onset, available idle surrogates that are ready to share their resources with the mobile device must be determined. Some researchers have addressed surrogate discovery in quite an impressive way.

2.2.2 Context gathering

To build a good decision about target execution location, there is a requirement to monitor available resources in surrogates and mobile devices and estimate application resource consumptions which is regarded as context gathering in some cyber foraging systems:

2.2.2.1 Partitioning

Here, a task is divided into smaller size subtasks, and undividable i.e. unmovable parts are specified. Some researchers do the partitioning automatically.

2.2.2.2 Scheduling

The most important step of cyber foraging is to assign individual task at the surrogate(s) or the mobile device most capable of executing it, based on the context information and the estimated cost of doing so.

2.2.3 Remote execution control

The final step includes the establishment of a reliable connection between the mobile device and the appropriate surrogate to pass required information, remote execution, and the receipt of returned results. Various researchers have considered remote execution control.

2.3 Cyber Foraging Goals

Cyber foraging is a way to execute resource intensive applications on resource constrained mobile devices. In fact, researchers in cyber foraging have tried to augment some resources of mobile devices in terms of effective metrics to obtain more efficient application execution. The most important resources have been considered by offloading approaches are as follows:

2.3.1 Energy

One of the most crucial limits of mobile devices is energy consumption because mobile device's energy cannot be replenished by itself. Many researchers have considered energy consumption as a factor for offloading.

2.3.2 Memory and storage

Memory capacity of mobile devices is less than stationary computers and memory intensive applications cannot usually run on mobile devices. Many researchers have considered the availability of memory and storage as another effective parameter for offloading decision.

2.3.3 Response time

When the processing power of mobile devices is considerably lower than static computers, task offloading is advantageous to decrease execution time. Many researchers that have considered the response time and latency as a major factor affecting the offloading decision.

2.3.3 I/O

Showing a movie on a bigger screen, playing music on more powerful speakers, and printing are examples of task offloading to improve I/O quality or exploit more I/O devices. Some researchers have focused on augmenting I/O as an effective parameter for offloading decision.

3. THE LOCUSTS FRAMEWORK

The LOCUSTS framework aims to provision developers with a complete cyber foraging toolbox that can ease the process of developing applications that utilize cyber foraging [6]. In the following the architecture of LOCUSTS will be briefly described in Section 3.1, and then the chosen approach towards task distribution is described in Section 3.2. Finally, task migration is illustrated in Section 3.3.

3.1 Architecture

A clear view of the current architecture of LOCUSTS system is shown in Figure 1. The LOCUSTS daemon runs as a different process on both client and surrogate devices and the individual applications can interact with the local LOCUSTS instance. As depicted, a cyber foraging enabled

Volume: 08 Issue: 06 Pages: 3274-3282 (2017) ISSN: 0975-0290

application includes some local code which is executed by the local device, and multiple number of distributable tasks. Every time a task is executed, the local LOCUSTS instance is communicated so that it may determine a suitable plan for execution. This execution plan is developed by the scheduler who depends on resource measurements, both local and remote.

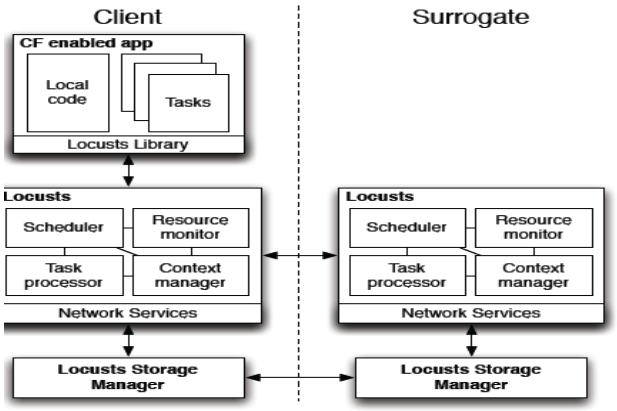

Fig. 1. LOCUSTS architecture

At the moment an execution plan has been obtained the task may either be distributed to one or more surrogates, or it may be transferred to the task processor for executing locally. At the bottom layer of the LOCUSTS client the network services layer exists. These help the mobile node to do all the required P2P operations like peer discovery, network roaming etc. All devices can opt to act as surrogates and thus the software running on surrogates and clients is identical. When performing operation as a surrogate, a device generally offers three things:

1) Execution of known tasks on behalf of clients
2) Client can author new tasks
3) Support for full task migration

The storage manager shown below the LOCUSTS daemon in above figure has a provision for a simple file system that can be accessed by tasks executed by LOCUSTS. The storage manager provides a virtual file system that can be accessed from within tasks. When executing a task on the local device, files in this virtual file system simply refer to the local files, but when a task is delegated for remote execution, the storage managers of the client and the surrogate are linked, so that remote files may be transparently perform read operation . This small distributed system is simple in design and is designed specifically for the purpose of cyber foraging. It has property of on-demand synchronization of file data to minimize the amount of data transferred, and has built in support for temporary files that will only be synchronized when the task is migrated.

3.2. Task Distribution

Task distribution is at the core of cyber foraging. The delegation of complex work to surrogates is the fundamental idea of cyber foraging. When designing a cyber foraging framework decision must be taken for exactly what is delegated, when it is delegated, and at which granularity. The question about task granularity is difficult to give a definite answer to, because it depends on numerous factors. The main factors to focus are network bandwidth and latency in network , processing power of the mobile device and the surrogate, the amount of energy used at the mobile device when communicating with the surrogate, and the rate of mobility of the device. When delegating tasks to a surrogate the mobile device needs to send the task to the surrogate, and similarly the surrogate must transmit a response back to the mobile device. This means that data must be transmitted over the wireless link between the mobile device and the surrogate. It must be taken care that whether the cost of this transmission, both in time and energy, is acceptable, i.e. whether the cost of distributing the task is smaller than the cost of doing the processing locally. Due to this reason distribution happens for only larger, longer running tasks, as the cost of delegating a small task will be more than the cost of local execution. But how does a small task get designated? It is variable as per context depending on factors mentioned above. To face this challenge, decisions regarding when to distribute a task is advisable to be taken dynamically depending on the current resource availability. This is achieved by resource usage monitoring at the client and surrogates and using this data in the scheduler application when future execution is being planned. The LOCUSTS framework takes the same approach towards task distribution; monitoring resource usage and dynamically deciding where and when to distribute tasks. In LOCUSTS task can be resized. A resizable task is a task that can be solved to different degrees which means that a surrogate may opt to solve only a small fraction of the task before returning the task to the client. It happens when the surrogate is under timing constrains given by the client. LOCUSTS also have functionality of the concept of migratable tasks as will be described in section below. The next important issue to face is exactly what is distributed and how it is done. When a mobile device is executing an application that can use cyber foraging, a part of the application will always be running locally while other parts may or may not be distributed to surrogates. The parts of the program that can be distributed must be identified and, possibly, modified to make the distribution possible. After identifying the parts of a program that can be delegated to surrogates a mechanism for actually distributing these tasks must be found. LOCUSTS do not have the need to preinstall anything on the surrogates. A task in LOCUSTS is therefore more than just an RPC invocation. It also contains the actual source code of the task presented in a way such that any surrogate, regardless of architecture etc., will be able to execute it. This means, that the portions of the code that designate distributable tasks must be written in a specific, interpreted language so that it can be moved on to surrogates, and thus allowing the clients to author new tasks on the surrogate. Allowing clients to execute unknown code on surrogates of course leads to an abundance of security issues that will have to be addressed, but that is out of the scope of this paper.

Volume: 08 Issue: 06 Pages: 3274-3282 (2017) ISSN: 0975-0290

Currently the language used for distributable tasks is Python.

3.3. Task Migration Distribution

Distributing very large tasks increases the merits of remote execution, since it helps to reduce the overhead of sending tasks back and forth. But, in existing cyber foraging frameworks, working with large tasks needs the user to stay within range of a specific surrogate for an extended period of time. To remove this problem, functionalities are made so that tasks may span multiple surrogates throughout their lifetime. The solution to the problem is task migration. Task migration enables surrogates to shift running tasks to other surrogates or even back on to the mobile device. Using migration a client no longer needs to stay within range of a surrogate while performing a task and it is thus possible to distribute larger tasks, which decreases the considerable overhead of remote execution. Task migration is depicted in Figure below. The ways that such task migration could be implemented range from simple surrogate-to-surrogate proxies to task check pointing. Both methods are used in LOCUSTS. Proxies are used in some scenarios when high speed network connections exist between surrogates. Take for example the scenario in Figure 2. The task is originated at S2 but when M moves out of range of S2 the task is migrated to S4. If, due to some reason, it makes sense to let S2 keep the task S4 will simply be asked to proxy for S2. In the viewpoint of the client M surrogate S4 is the one executing the task and all communications about the task goes through S4. Alternatively, the task could be moved entirely to S4. Subsequently the current executing task would be check pointed by S2 and its code and state sent to S4.

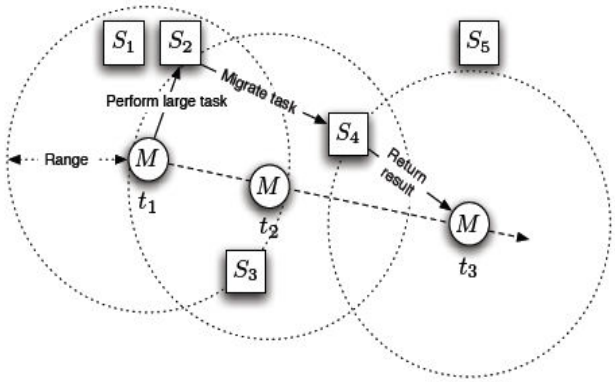

Fig. 2 Task migration in LOCUSTS

Many factors must be considered when choosing which kind of migration to use – factors such as network bandwidth between the surrogates, current resource usage at the surrogates, checkpoint size, estimated finish time of the task etc. This preceding description of task migration touches lightly on a very complex matter.

3.4 Slingshot

Slingshot is a new architecture for deploying mobile services at wireless hotspots[8]. Slingshot replicates applications on surrogate computers which are 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 which is 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). Additionally, the first-class replica is a trusted repository for application state, which means upon surrogate failure state is not lost but it is preserved. Slingshot also makes surrogate management easy. It considers virtual machine encapsulation to remove the need to install application-specific code on surrogates., Replication prevents the loss of application state when a surrogate crashes or even permanently fails. The performance impact of surrogate failure is reduced by other replicas, which continue to service client requests. The harnessing of surrogate computation is a complex problem with many challenges. Our paper addresses these challenges, including improving interactive response time, hiding the perceived cost of migration, recovering from surrogate failure, and simplifying surrogate management. It also gives robust results that measure the substantial benefit of surrogate computation for stateless and stateful applications. Other research 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. Current implementation of Slingshot has two services: a speech recognizer and a remote desktop. Observatory results show that instantiating a second-class replica on a surrogate lets these applications run nearly 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.

4.1 DESIGN PRINCIPLES

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

4.1.1 Location

Server location can be crucial to the performance of remote execution. Suppose a handheld device is connected to the Internet at a wireless hotspot. If the handheld device 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 general 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 reduce 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.

Volume: 08 Issue: 06 Pages: 3274-3282 (2017) ISSN: 0975-0290

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 unexpected changes of Internet routing. Firewalls, VPNs, and NAT components add additional latency when connections cross administrative boundaries [9]. For mobile users, a journey of only a few hundred feet can enhance the round-trip time for communication with a remote server. But a server located at the current hotspot is only a network hop away.

4.1.2 Replicate instead of migrate

The need to locate services near mobile users refers that services need to move over time. When a handheld user moves to a new location, a surrogate at the new hotspot will generally offer better response time than a surrogate at the previous hotspot. In case we need to move functionality, one option is migration: suspend the application on the previous surrogate, send its state to the new surrogate, and resume it there. This approach affects the availability of the application while it is migrating. Slingshot deploys an alternative strategy that instantiates multiple replicas of each service. During instantiation of new replicas, 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[10]. 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. Slingshot instantiates a new replica by check pointing 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.

4.1.3 Ease of maintenance

We observe that the business case for deploying a surrogate as being similar to that of deploying a wireless access point. Desktop computers have become cheaper, not much more than an access point. Further, it has been shown by researchers that surrogates can provide significant value-addition to wireless users in terms of improved interactive performance. Nevertheless, 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 need negligible or no supervision. They should be appliances that require little configuration. Apart from that maximum trouble shooting should be possible by a normal restart. For easy management of surrogates Slingshot has the following policies**.**

4.1.3.1 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.

4.1.3.2 Uses a heavyweight virtual machine

While Para virtualization and other lightweight approaches to virtualization offer scalability and performance benefits, they also restrict the type of applications that can run within a VM. In contrast, by deploying a heavyweight VMM (VMware), Slingshot runs the two applications described in Section 4.2 without modifying source code, even though their guest OS (Windows XP) differs substantially from the surrogate host OS (Linux)[11].

4.1.3.3 Places no hard state on surrogates

As surrogates have only soft state, a normal or abnormal restart does not result to incorrect application behavior or data loss. If a surrogate crashes or is restarted, the only effect observable by the user is that performance goes down to the level that would have been available in the absence of the surrogate.

4.2 Slingshot implementation

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 partitioning that fit this model. For some applications, the best partitioning may not be immediately clear—in these cases, we could leverage prior work to choose a partition that fits our model. In Figure 1, a first-class replica executes on the home server and a second-

Volume: 08 Issue: 06 Pages: 3274-3282 (2017) ISSN: 0975-0290

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. 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 image of the virtual machine, and the volatile state, which includes its memory image and registers. On the home server, requests are redirected 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.

4.3 Slingshot applications

We have adapted the IBM Via Voice speech recognizer and the VNC remote desktop to use Slingshot[5]. 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.

4.3.1 Speech recognition

We chose speech recognition as our first service because of its natural application to handheld computers. We used IBM Via Voice in our work. We created a server side proxy that accepts audio input from a remote client and passes it to Via Voice through that application's API. Via Voice returns a text string which the proxy sends to the client. Via Voice 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[12]. 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 quiescence 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 nondeterministic. 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. 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

Volume: 08 Issue: 06 Pages: 3274-3282 (2017) ISSN: 0975-0290

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[13]. Trusted computing efforts 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.

5. PUPPETEER

5.1 Introduction

Puppeteer is advocated as a system for using applications based on components in mobile environments. It reaps the advantage of the exported interfaces of these applications and the structured nature of the documents they manipulate to perform adaptation without changing the applications. The system is structured in a modular fashion, allowing easy addition of new applications and adaptation policies. The initial prototype concentrated on adaptation to limited bandwidth. It was designed to run on Windows NT, and included support for a variety of adaptation policies for Microsoft PowerPoint and Internet Explorer 5[7]. We represent in this paper that Puppeteer can support complex policies without any major change to the application and with negligible cost.

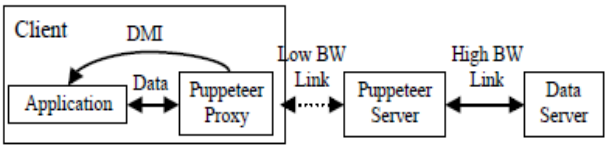

Fig 3 Overall architecture

Figure above shows the four-tier Puppeteer system architecture. It includes the applications which are to be adapted, the Puppeteer client proxy, the Puppeteer server proxy, and the data server. The application and data server are completely untouched. The Puppeteer client proxy and server proxy work combinely to perform the adaptation. The Puppeteer client proxy is in control of executing the policies that adapt the applications. The Puppeteer server proxy has to parse the documents, exposing their structure and transcoding components as needed and requested by the client proxy. The Puppeteer server proxy is considered to have robust connection with the data server. In the most common situation, it executes on the same machine as the data server. Data servers can be random repositories of data such as Web servers, file servers or databases.

5.2 Puppeteer Architecture

The Puppeteer architecture is represented by four types of modules: Kernel, Driver, Transcoder, and Policy (see Figure 2). The Kernel appears once in both the client and server Puppeteer proxy. A driver supports adaptation for a particular elemental type. A driver for a specific elemental type may call on a driver for another elemental type, if an elemental of the modern type is included in an elemental of the basic type. The driver for a specific application resides on the top of this driver hierarchy. Driver execution may happen in both the client and the server Puppeteer proxies, as may Transcoders which model particular transformations on elemental types. Policies denote specific adaptation mechanisms and subsequently execute in the client Puppeteer proxy.

5.2.1 Kernel

The Kernel is a part which is independent module that derives the Puppeteer protocol. The Kernel executes in both the client and server proxies and triggers the shift of document components. The Kernel does not include knowledge regarding the specifics of the documents being transmitted. It functions on a representation neutral description of the documents, which are referred to as the Puppeteer Intermediate Format (PIF). A PIF has a skeleton of components, each of which possess a set of related data items. The skeleton includes the structure of the data used by the application. it has the form of a tree, with the root being represented as the document, and the children being pages, slides or any other elements in the document. The skeleton is a multi-level data structure as elements in any level can contain sub elements. The skeleton is element independent, but elements in the skeleton are element specific[8]. To improve functional performance, the Kernel batches requests for multiple elements into a single message and has provisional support for asynchronous requests.

5.2.2 Drivers

Puppeteer requires an import and an export driver. A tracking driver is necessary to implement compound policies. The import drivers resolves the documents, extracting their elemental structure and converting them from their application limited file formats to PIF.In the general case where the application's file format is parsable, either because it is human readable (e.g., XML) or there is sufficient documentation to create a parser, Puppeteer can parse the files directly to uncover the structure of the data[13]. This leads to good performance, and enables clients and server to run on different platforms. When the application only exposes a DMI, but has an opaque file format, Puppeteer runs an particular of

Volume: 08 Issue: 06 Pages: 3274-3282 (2017) ISSN: 0975-0290

the application on the server, and handling the DMI to uncover the structure of the data, in some cases using the application as a parser. This configuration allows for a high degree of flexibility, since Puppeteer need not understand the application's file format. It creates, however, more overhead on the server proxy, and requires both the client and server to run the environment of the application, which in most cases amounts to running the same operating system on both servers and clients. resolving at the server does not work well for documents that choose what data to fetch and display by executing a script. Instead, import drivers for dynamic content run in the Puppeteer client proxy, and built on a prevent mechanism that traces requests. Regardless of whether the skeleton is built statically in the server proxy or dynamically in the client proxy, any changes to the skeleton are reflected by the Kernel at both ends to maintain a persistent view of the skeleton. Export drivers unparsed the PIF and update the application using the DMI interfaces exposed by the application. A minimal export driver has to support inserting new components into a running application. Tracking drivers are important for many complex methods. A tracking driver tracks which components are being viewed by the user and intercepts load and save requests. Tracking drivers can be implemented using polling or event registration mechanisms.

5.2.3 Transcoders

Puppeteer makes huge use of transcoding to perform conversions on basic data. Transcoders include the conventional ones, such as compression and diminish image resolution. An exclusive transcoding system is used to setup loading subsets of components. Each element of the PIF skeleton has a number of associated data items that, among other things, encode in a component-distinct pattern the relationship between the component and its children. To load a subset of the children of a given node, it is required to change the data items associated with the parent node to follow the fact that we are only loading some of its children. In effect, by transcoding the parent node's data items, we create a new temporary component that consists only of a subset of the children of the original component.

5.2.4 Policies

Policies are modules that work on the client proxy and control the yielding of components. These policies traverse the skeleton, choosing what components to fetch and with what integrity. Puppeteer provides support for two types of Policies: general purpose Policies that are independent of the component type being adapted and component define Policies that use their knowledge about the component to drive the adaptation. Typical Policies choose components and integrities based on available bandwidth and user defined preferences. Other Policies track the user or react to the way the user moves through the document. Regardless of whether the decision to fetch a component is made by a general purpose Policy or by a component specific one, the actual data transfer is performed by the Kernel, free the policy from the elaborateness of communication.

5.3 The Adaptation Process

The adaptation process in Puppeteer is divided into three stages: resolving the document to uncover the structure of the data, fetching the initially selected components at distinct integrity levels and supplying those to the application, and, if the policy so defines, modernizing the application with newly fetched data. When the user opens a document, the Kernel on the Puppeteer server proxy instantiates an import driver for the proper document type. The import driver resolves the document, extracts its skeleton and data, and generates a PIF. The Kernel then transfers the document's skeleton to the Puppeteer client proxy. The policies running on the client proxy ask the Kernel to fetch an initial set of components at a distinct integrity. This set of components is supplied to the application in return to its open call. The application has finished loading the document, returns control to the user. Meanwhile, Puppeteer knows that only a portion of the document has been loaded. The policies in the client proxy now decide what further components to fetch. They instruct the Kernel to do so, and then the client proxy uses the DMI to feed those newly fetched components to the application.

6. CONCLUSION

As we have discussed in this paper the cyber foraging technique is becoming popular day by day due to increased mobility of device by users. Users want a cost effective way of computing needs and cyber foraging provides just the exact functionality. It has been observed that many applications similar to the systems like LOCUSTS, Slingshot, and Pupetter are coming up in near future which will be released for commercial use in couple of years in a massive way. The popularity of this application is going to surge in an upward direction in years to come. Researchers in this area are actively working to realize this.

REFERENCES

[1] Perry,Mark, O'hara,Kenton, Sellen,Abigail, Brown,Barry&Harper,Richard, (2001) "Dealing with Mobility: Understanding Access Anytime, Anywhere,"*Transactions on Computer-Human Interaction (TOCHI), Vol.* 8, No. 4, pp. 323-347.

[2] M. Satyanarayanan, (2001) "Pervasive Computing: Vision and Challenges," IEEE Personal Communication, Vol. 8, No. 4, pp. 10-17..

[3] Balan, Rajesh Krishna, Flinn, Jason, Satyanarayanan, Mahadev, Sinnamohideen,S.&Yang, H.I.,(2002) "The Case for Cybef Foraging," presented at the 10th Workshop on ACM SIGOPS European Workshop: beyond the PC, New York, NY, USA, pp. 87-92..

Volume: 08 Issue: 06 Pages: 3274-3282 (2017) ISSN: 0975-0290

[4] Chun, Byung-Gon & Maniatis, Petros, (2010) "Dynamically Partitioning Applications between Weak Devices and Clouds," in 1st ACM Workshop on Mobile Cloud Computing and Services(MCS 2010), San Francisco, pp. 1-5.

[5] Kemp, Roelof, Palmer, Nicholas, Kielmann, Thilo, Seinstra, Frank, Drost, Niels, Maassen,Jason&Bal, Henri, (2009) "eyeDentify: Multimedia Cyber Foraging from a Smartphone," in IEEE International Symposium on Multimedia (ISM2009), San Diego, pp. 392-399.

[6] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to network and client variability via on-demand dynamic distillation. Sigplan Notices, 31(9):160–170, September 1996.

[7] Randy H. Katz. Adaptation and mobility in wireless information systems. IEEE Personal Communications, 1(1):6–17, 1994.

[8] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. Operating Systems Review (ACM), 51(5):276–287, December 1997.

[9] M. Satyanarayanan, (2001) "Pervasive Computing: Vision and Challenges," IEEE Personal Communication, Vol. 8, No. 4, pp. 10-17.

[10] Balan, Rajesh Krishna, Flinn, Jason, Satyanarayanan, Mahadev, Sinnamohideen,S.&Yang, H.I., (2002) "The Case for Cybef Foraging," presented at the 10th Workshop on ACM SIGOPS European Workshop: beyond the PC, New York, NY, USA, pp. 87-92.

[11] Balan, Rajesh Krishna, Satyanarayanan, Mahadev, Park, SoYoung&Okoshi, Tadashi,(2003) "Tactics-Based Remote Execution for Mobile Computing," in 1st International Conference on Mobile Systems, Applications and Services, San Francisco, pp. 273-286.

[12] Gu, Xiaohui, Messer, Alan, Greenbergx, Ira, Milojicic, Dejan&Nahrstedt, Klara, (2004) "Adaptive Offloading for Pervasive Computing," IEEE Pervasive Computing Magazine, Vol. 3, No. 3, pp. 66-73.

[13] Ou, Shumao, Yang, Kun&Liotta, Antonio,(2006) "An Adaptive Multi-Constraint Partitioning Algorithm for Offloading in Pervasive Systems," in 4th Annual IEEE International Conference on Pervasive Computing and Communications (PERCOM'06), Pisa, Italy, pp. 116-125.