A Programming Interface for Application-Aware Adaptation in Mobile Computing

Brian D. Noble, Morgan Price, and M. Satyanarayanan

February 1995

CMU-CS-95-119

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

To appear in the Proceedings for the Second USENIX Symposium on Mobile and Location-Independent Computing, April 1995, Ann Arbor, Michigan.

Abstract

Mobile clients face wide variations in network conditions and local resource availability when accessing remote data. Coping with this uncertainty requires the ability to retrieve and present data at varying degrees of *fidelity*. In this paper we present *applicaton-aware adaptation* as a solution to this problem. The essence of our solution is a collaborative partnership between applications and the operating system. We describe the *Odyssey* API for application-aware adaptation and demonstrate its use in accessing two types of data: video and maps.

This research was supported by the Air Force Materiel Command (AFMC) and the Advanced Research Projects Agency (ARPA) under contract number F19628-93-C-0193. Additional support was provided by the IBM Corporation, Intel Corporation and AT&T Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFMC or ARPA or the U. S. Government, nor of any other supporting agency.

Keywords: adaptation, mobile computing, distributed systems, video data, spatial data, UNIX

1. Introduction

Mobile clients face many challenges in accessing data from servers. Because a mobile client has to be compact and lightweight, it is typically resource-poor relative to a desktop client. Network connectivity, especially via wireless media over a large area, tends to vary considerably in bandwidth, latency, reliability and cost. Power management considerations often require certain actions to be deferred, avoided or slowed down to prolong battery life. The relative costs of accessing distributed services changes as mobile clients move. Finally, the very nature of mobility has a negative impact on robustness and security.

As a consequence of these constraints, the mechanism for mobile data access has to be *adaptive* in nature, dynamically conforming to the limitations of individual clients and their current environments. We believe that such adaptation can best be performed by a collaborative partnership between the operating system and individual applications. We refer to this strategy as *application-aware adaptation*[10].

Application-aware adaptation characterizes the design space between two extremes. At one extreme, adaptivity is entirely the responsibility of individual applications. This means that there is no focal point in the system to resolve the potentially incompatible resource demands of individual applications. It also means that there is no way to enforce limits on resource usage. At the other extreme, adaptivity is completely subsumed by the system. Although the feasibility of this approach has been demonstrated in systems such as *Coda*[5, 9], there are limits to its applicability. In particular, the end-to-end argument[8] suggests that there will be circumstances where only an application can determine the best form of adaptation. Unless the system is extended to incorporate specific knowledge about every application, there will be situations where adaptation by the system will be inadequate or even counter-productive. By striking a balance between these extremes, application-aware adaptation offers a more promising approach to mobile data access. It permits individual applications to determine how best to adapt, but allows the system to retain management of key resources and enforcement of decisions regarding their usage.

How can application-aware adaptation be effectively supported? This paper is a status report on our work toward answering this question. This work is being done in the the context of *Odyssey*, an experimental Unix platform for mobility. We have implemented a preliminary prototype and have demonstrated its use in two applications accessing data in a mobile environment. While rudimentary in many respects, our prototype does provide initial evidence of the feasibility and effectiveness of application-aware adaptation.

We begin the paper by introducing the concept of *data fidelity* and discussing the central role it plays in application-aware adaptation. Next, we discuss a number of factors influencing our design. We then describe the design of Odyssey, focusing specifically on its support for application-aware adaptation. Finally, we describe the implementation and status of our prototype.

2. Data Fidelity

Under ideal circumstances, the data presented at a mobile client should be identical to the current server copy. As resources become scarce, it may no longer be feasible to completely preserve this correspondence; some form of degradation is unavoidable. How does one characterize the extent of this degradation? We define *fidelity* as the degree to which a copy of data presented for use matches the reference copy.

Fidelity has many dimensions. One well-known, universal dimension is consistency. Other dimensions depend on the type of data in question. For example, video data has at least two additional dimensions: frame rate and image quality of individual frames. Spatial data, such as topographical maps, have dimensions of minimum feature size or resolution. For telemetry data, appropriate dimensions include sampling rate and currency.

The dimensions of fidelity are natural axes of adaptation for mobility. But the adaptation cannot be solely determined by the type of data; it also depends on the application. As we show in the next section, different applications using the same data may make different tradeoffs among dimensions of fidelity.

2.1. Video Data in Mobile Environments

Consider a movie stored on a server, and two applications accessing that video stream from a mobile client. The first application is a video playback application, *player*, and the second, *editor*, is a video scene editor. These two applications must make different fidelity tradeoffs in accessing the same video stream. No single policy can satisfy them both.

The player's primary goal is to preserve correspondence between movie time and real time. A secondary goal is to play the movie at the original frame rate, resolution, and image quality. In times of plentiful resources, the player can indeed meet both goals. However, when network bandwidth becomes scarce, the player may have to sacrifice its secondary goal in order to meet its primary goal. Thus, it may choose to switch to a black-and-white stream at full frame rate, to drop frames, or otherwise reduce the bandwidth requirements of the stream. To guard against total disconnection, the player may even hoard a very low quality version of the movie.

The editor's main goal is very different from that of the player; it must ensure that the user sees every frame of the video stream to allow precise editing. To allow this, the editor is willing to relax the correspondence between movie time and real time. Thus, when network bandwidth decreases, the editor will access the movie at a rate slower than real time to avoid dropping frames.

It is hard to see how any single operating system policy can adequately service both of these applications' needs, even though they are accessing exactly the same data. Regardless of the system's decision to change the fidelity of the stream it is retrieving, either the player or the editor – and quite possibly both – will not be satisfied. No system can be clever enough to anticipate and satisfy every application's needs. On mobile machines, where the environment is unpredictable, such unsatisfactory service will be even more evident.

3. Design Considerations

What is required to support application-aware adaptation? Generally, the system must provide a set of API extensions that allow applications to track and react to their environment, and a system architecture which effectively supports these extensions. In the sections below, we outline the desired properties of the API extensions and supporting architecture.

3.1. API Extensions

In order for applications to make decisions based on their environment, they must be able to name aspects of the environment that are important to them. This naming mechanism must be both simple and extensible. Applications should be able to specify exactly those features of the environment in which they are interested, and be notified of changes to just those features. Such specification and notification should be efficient. They must also fit into the programming style and culture of the base operating system, but cannot depend on esoteric features. Popular applications run on an increasingly diverse set of operating systems; providing common adaptation facilities enhances the portability of such applications.

As applications track changes in the environment, they must adapt their access to data. Some types of adaptation will require changes in operating system policy. There must be an efficient, flexible, and extensible mechanism to request such changes. Since the operating system is the final arbiter of resource usage, the request need not always be honored.

Resource	Units	Reference Item?
Network Bandwidth	bits per second	yes
Network Latency	microseconds	yes
Disk Cache Space	kilobytes	no
CPU	SPECints available	no
Power	minutes of computation	no
Money	cents	no

This figure lists the generic resources defined for the Odyssey system. The first column lists the name of the resource. The second column gives the units in which the resource is measured. The third column specifies whether or not the resource is measured with respect to a particular item in the Odyssey store. Of particular interest is the last item, *money*. Many experimental implementations of electronic money as well as systems that use money in exchange for services exist. We believe that such services, particularly those which offer some sort of query facility, will become more common. Note that these are only the generic resources; there may be others that are type-specific.

Figure 1: Generic Resources in Odyssey

3.2. Supporting Architecture

What of the underlying architecture supporting these extensions? The overriding goal is *simplicity*. We are not trying to invent a new operating system, but merely extend existing ones in simple ways. We have striven to keep such extensions minimal, while making them powerful enough to explore application-aware adaptation for a wide range of data types.

It is important to note that we do not attempt to provide *resource guarantees* to applications. Such guarantees, typically encountered in real-time systems, require guarantees from lower layers of the system. But the environment of a mobile computer is too unpredictable for such guarantees. Hence, we only promise to inform applications when their environment changes, and arbitrate between applications competing for scarce and unpredictable resources.

Finally, our architecture should adhere to sound principles of software engineering. Some functionality in support of the API will be independent of the type of data, while other functionality will be type-specific. The architecture should provide isolation between different types of data as well as between the generic and type-specific portions of the system.

4. Odyssey API

This section describes our design of the Odyssey API supporting application-aware adaptation. As discussed above, there are three components to the Odyssey API. First, there is a way for applications and the system to talk about salient features of the environment. Second, there is a mechanism that enables applications to track their environment. Third, there is a mechanism through which applications request policy changes based upon their environment.

4.1. What is an Application's Environment?

We consider the salient features of an application's environment to be the *resources* available to that application. Such resources can be either *generic* or *type-specific*. Generic resources have meaning for all items stored in Odyssey. Examples of generic resources include network bandwidth between the mobile client and the server storing an item, available disk space on the mobile client, and battery power remaining on the mobile client. The generic resources in Odyssey are listed in Figure 1.

Type-specific resources have meaning only for items of a particular type. For example, consider a commercial database that indexes items in the World Wide Web. Such a service might sell a *subscription* that enables a client to make some number of queries per day. The number of queries left in a given day is a resource that is sensible only in the context of queries against that database.

Odyssey tracks and reports the *availability* of a resource, and how that availability changes. We measure the availability of an individual resource with a single scalar value. The units of a particular resource's availability are chosen appropriately for that resource. For example, network bandwidth is measured in bits per second. Available disk space is measured in kilobytes. Power remaining to a laptop is measured in minutes of operation.

Some resources are estimated with respect to a particular item in the Odyssey store. We call such items *reference items*. For example, network bandwidth between a mobile client and a server differs for different servers. Thus, we only speak of network bandwidth with respect to a particular reference item; the bandwidth in question is that between the client and the server storing that particular item. Since type-specific resources only have meaning for items of a particular type, they always have reference items.

4.2. How to Track the Environment?

For an application to track the availability of resources two things must happen. First, the application must inform the system of the resources in which it is interested. Second, the system must monitor the availability of resources, and notify the application when the availability of one or more relevant resources changes in an interesting way. For efficiency, we chose to use asynchronous notification rather than polling in Odyssey.

Naturally, not all applications will be interested in the same set of resources. To tell the system what resources an application is interested in, the Odyssey API provides a call, `ody_request`. For example, an application making an `ody_request` might ask, "Please invoke procedure `bar` if the network bandwidth between here and the server storing `/ody/foo.c` exceeds ten Mb/s or falls below one Mb/s." The C declarations for `ody_request` and associated data structures appear in Figures 2 – 4.

Requests name the *resource* of interest, the *bounds of tolerance* on that resource's availability, the *reference item*, and an *upcall procedure*. In our example above, the resource of interest is network bandwidth. The upper tolerance bound is ten Mb/s, and the lower bound is one Mb/s. The reference item is `/ody/foo.c`, and the upcall procedure is the procedure `bar`.

The resource is named in the `ody_req_des_t` structure, as are the tolerance bounds and the address of the upcall procedure, which is a handler function much like a signal handler. The resource is named by an integer identifier. Generic resource identifiers are known throughout the system; type-specific identifiers are known only to portions of the system that implement that type, but are limited to a specific range. If the resource is not within the specified tolerance bounds, the call fails and returns the current value in `res`. Otherwise, the request is registered with the system. If the resource later strays outside of those bounds, the system invokes the handler through an upcall.

If a request is granted by the system, the system returns a request identifier. That request identifier is also passed to the request handler when the application is notified by the system. If the application no longer wishes to be notified for that request, it can invoke `ody_cancel` on it.

4.3. How to Request Policy Change?

As applications are notified of resource changes, they will need to adapt their access patterns. Some of this adaptation will require changes in policy within the operating system. Since policies are type-specific, these requests for changes in policy must also by type-specific. We call such a request a *type-specific operation*, or `ody_tsop`. An example of a type specific operation would be, "Please switch from the full color version of this stream to the black-and-white version."

```
/* Pathname resource request */
int ody_request (path, req, res);
char            *path;          /* pathname of reference item */
ody_req_des_t   *req;           /* A request descriptor */
long            *res;           /* The request id returned, or current value */

/* Cancel a request */
int ody_cancel (reqid);
long reqid;             /* The request to cancel */
```

This figure shows the C declarations for the pathname-based version of `ody_request`, as well as `ody_cancel`. The descriptor-based version is identical except that a file descriptor is used instead of `path`. Note that `ody_request` is similar to the UNIX `sigvec` system call. `ody_request` allows an application to place a notification request `req`; `ody_cancel` cancels an outstanding request. Declarations for relevant data structures can be found in Figure 3; the signature for the callback function to be invoked on notification of an outstanding `ody_request` is shown in Figure 4.

Figure 2: C Declaration for `ody_request` and `ody_cancel`

```
/* A version stamp*/
typedef struct {
    long        gs;         /* Version of generic resource interface*/
    ody_codex_t codex;      /* The type of the reference item*/
    long        cs;         /* Version of type-specific resource interface*/
} ody_vers_t;

/* A resource request descriptor */
typedef struct {
    long            resource;       /* Resource identifier */
    ody_vers_t      version;        /* Version stamp */
    long            low, high;      /* low, high values of window */
    ODY_REQ_FN_T    fn_ptr;         /* function to call if window is left */
} ody_req_des_t;
```

These are the principal data structures used in the `ody_request` call. `ody_vers_t` is used to ensure that the application and system are using the same set of resource identifiers, and that the application and the system agree on the type of the reference item. The type `ody_codex_t` is an enumeration of known types in the system, called *codices*. The `req_des_t` type holds the fields of a request: the resource, version information, the window of tolerance, and the upcall procedure. The signature for upcall procedures is shown in Figure 4.

Figure 3: Data Structures for `ody_request`

```
/* A resource request handler */
typedef void (*ODY_REQ_FN_T)(long, long, long);
/* the three arguments are: */
/*     the request id to which this notification is responding */
/*     the resource identifier */
/*     the current value of the resource */
```

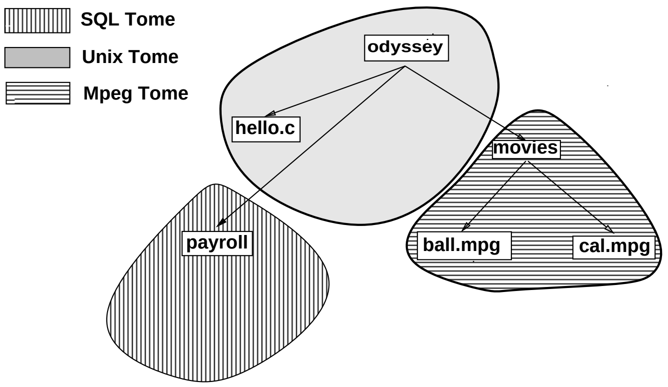
This figure shows the type signature of a request handler. A request handler takes three arguments: the request identifier, as returned by `ody_request`, to which this notification is responding, a resource identifier denoting the resource that has changed, and the new availability of that resource.

Figure 4: Notification Handler Declaration

```
/* Pathname-based type specific operation */
long ody_tsop (path, vers, op, argsz, arg, retsz, ret);
char        *path;     /* pathname of reference item */
ody_vers_t   vers;     /* version of this codex' interface */
long         op;       /* which operation to perform */
size_t       argsz;    /* size of argument buffer */
void        *arg;      /* arguments for operation */
size_t       retsz;    /* size of return buffer */
void        *ret;      /* return buffer */
```

This figure shows the C declaration for `ody_tsop`, the pathname-based invocation of a type-specific operation. The descriptor-based version is identical except that a file descriptor is used instead of `path`. The arguments name the reference item, version information, the operation to be performed, and buffers for the arguments and results. The definition of `ody_vers_t` can be found in Figure 3. The sizes of the argument and result buffers must be passed, so that layers that do not know the details of the particular type can pass arguments correctly. Note that this is similar in flavor to the UNIX `ioctl` system call.

Figure 5: C Declaration for `ody_tsop`



This figure illustrates a sample Odyssey namespace. In this example, there are three *tomes*, each of a different type. The first tome, rooted at `odyssey`, contains the single UNIX file `hello.c`. The second, rooted at `payroll`, is a database. Note that no nodes appear inside of `payroll`; it is named associatively rather than hierarchically. The third tome, rooted at `movies`, contains two MPEG movies, `ball.mpg` and `cal.mpg`.

Figure 6: Odyssey Tomes

Just as there is no way to predict the needs of all applications , there is also no way to predict all possible requests for policy changes. So, instead of trying to enumerate them for each type *a priori*, we instead provide a general mechanism to allow for experimentation and extension. The C declaration for `ody_tsop` appears in Figure 5.

To invoke `ody_tsop`, an application must specify a reference item. It must also specify the operation to perform, the arguments to the operation, and a buffer for the return value. The type of the reference item determines the type of the `ody_tsop`, and the reference item is passed through to the body of code that implements the `ody_tsop`. The reference item can be specified by file descriptor or pathname. The operation is denoted by an integer identifier, and need only be unique within a single type, thus preserving independence between different types. The sizes of these buffers are specific to the operation; hence, they are also type-specific. To preserve isolation between generic and type-specific portions of Odyssey, the sizes of these buffers must be specified.

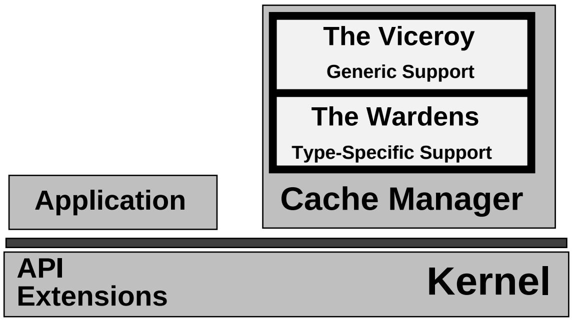
The type-specific operation mechanism is designed to allow applications to make policy requests. However, once it is present, `ody_tsop` can be leveraged to provide a set of access methods richer than the simple file system interface provided by common operating systems. For example, items of type "video" might support the type-specific operation `video_read_frame`, which reads a single variable sized frame, in addition to the simpler `read` system call. Such extension allows us to use data of different types in ways that are natural to the data, rather than forcing the data to fit the more restrictive file system model.

5. Odyssey Structure

To support the Odyssey API, our design provides three extensions to UNIX. First, we have added a notion of *type* to the standard UNIX file system. Second, we have added a generic cache manager, the *viceroy*, to provide type-independent support for the Odyssey API. Third, we have provided a set of *wardens*, which are part of the Odyssey cache manager, each providing support for an individual type in the Odyssey store. The next three sections explore each of these in turn.

5.1. Adding Types to the Operating System

Odyssey provides a single, global namespace to its clients. A simple example of such a namespace is shown in



This figure illustrates the architecture of an Odyssey client. Odyssey applications make use of the Odyssey API extensions along with the operating system's API. Operations on Odyssey objects are redirected by the kernel to the cache manager, which is at user level for ease of implementation. The cache manager is split into two logical pieces: the viceroy, providing generic support, and a set of wardens, each supporting a single type.

Figure 7: Odyssey Client Architecture

Figure 6. This namespace is broken into subspaces called *tomes*, or *typed volumes*. Tomes are similar to *volumes* in AFS and Coda[11, 4, 9]. A tome carries with it a notion of type; all items in a tome are of the same type. A tome's type determines type-specific resources, operations, and dimensions of fidelity for items in that tome. All tomes which have the same type are logically grouped together into a *codex*.

We envision a small number of types in Odyssey. The implementation effort to add a type is nontrivial, and will likely be undertaken by experienced system builders. A new type will be justified when applications using data of that type exhibit access patterns fundamentally different from any other existing ones. In the video example in Section 2.1, the player and editor have roughly the same access patterns, but prefer to make different tradeoffs. In contrast, video data, which is inherently linear, will be accessed differently from topographical maps, which are inherently spatial.

5.2. Providing Generic Support

There are many client-side tasks that are independent of data type. This generic functionality is implemented by the *viceroy*. The viceroy can be thought of as the generic cache manager, which depends on type-specific cache managers to do its job. The logical role of the *viceroy* is illustrated in Figure 7.

The viceroy's most important task is to act as the single point of resource control in the system; all other pieces of the Odyssey client are subordinate to it. The viceroy also handles requests for generic resources, and notifies applications when those resources leave requested bounds. Finally the viceroy responds to requests on individual Odyssey objects, and forwards them to the appropriate warden.

5.3. Providing Type-Specific Support

We call Odyssey's type-specific cache managers *wardens*. There is one warden in the Odyssey cache manager for each type in the Odyssey store. The wardens' logical role on the client is illustrated in Figure 7.

The wardens are responsible for implementing the access methods on objects of their type – both the standard UNIX operations as well as the type-specific ones. The wardens also implement a number of different fidelity mechanisms, and allow applications to choose between them. In addition, they provide reasonable default policies for naive applications. Default policies are also important in providing backward compatibility with legacy applications.

6. Implementation Status

We have built a preliminary prototype of the Odyssey client along with applications, wardens and servers for two data types. The goals of the prototype were twofold. First, we wanted to code applications that might benefit from application-aware adaptation to the Odyssey API to test the efficacy of the interface. Second, we wished to explore the practical implications of the division between viceroy and warden.

The two data types we have explored are QuickTime[1] and GRASS[7]. QuickTime is a multi-media encoding standard proposed by Apple Computer. GRASS is a public domain geographical information system. Along with some basic applications using these data types, we provide a simple control program to a user of the prototype. The control program is used to simulate various network bandwidths on the connection between the cache manager and various servers. The applications then change the fidelity of the data they access to match the simulated bandwidth. While each application works well in isolation, we have not yet explored resource control mechanisms to arbitrate between them.

The QuickTime application we have explored is a movie player. The player can open a QuickTime movie on a server via the Odyssey cache manager and begin playing it. The server stores the movie at several different levels of fidelity, and bundles them into a logical movie. The player, by using `ody_request` and responding to notifications, asks the cache manager to fetch the highest fidelity stream that can be played in real time given the available bandwidth.

The GRASS prototype supports applications via a modification to the GIS library. These applications display, query, and combine geographical data. The main type of data is raster data: a two-dimensional array of values set into a coordinate space. The client caches files from the server in the local filesystem; the raster data is fetched at various resolutions, depending on available network bandwidth. The GRASS applications then access those cached files.

We have made many simplifications for ease of rapid prototyping. The current prototype is completely user-level, trading realistic resource management policies and performance for simple implementation. It makes no attempt to measure resources, and depends on the control program instead. The UNIX file system call interface is not currently implemented; the application uses the Odyssey API exclusively in communicating with the viceroy, and uses the local file system when necessary for a cache. The prototype consists of a library linked into Odyssey applications, a prototype cache manager and wardens, and the applications and servers. Each of these is discussed below.

6.1. The Odyssey Library

The API extensions are provided by a library linked with the prototype application. All of the calls described in Section 4 are provided, but the prototype does not include the standard file system interface. The library communicates with the cache manager via RPC. The library responds to all notifications by the prototype cache manager, and forwards them to the proper upcall handler registered by the application; the UNIX `signal` interface is used to simulate upcalls.

6.2. The Prototype Cache Manager

The prototype cache manger consists of a simple viceroy, along with the QuickTime and GRASS wardens. It performs minimal resource management, and makes no attempt to authenticate users or arbitrate between conflicting applications. Rather than attempt to estimate resources, it depends on the external control program to advise it. It implements `ody_request` and `ody_cancel`, and forwards `ody_tsop` operations to the wardens based on the reference item's type. It notifies applications by sending them a signal, and passing information about the notification through the filesystem. In the sections below, we describe both the QuickTime and GRASS wardens.

`QT_OpenMovie(m)`	Open movie `m` and return track information.
`QT_CloseMovie(m)`	Close movie `m` and free resources.
`QT_GetFrame(t)`	Returns the first frame to display after time `t`.
`QT_SwitchTracks(m,i)`	Ask to make track `i` of movie `m` the active track

Figure 8: Operations Supported by the QuickTime Warden

6.2.1. The QuickTime Warden

The QuickTime warden exports the interface we envision for the final system. It has no type-specific resources, but has four type-specific operations. Those operations are shown in Figure 8. `QT_SwitchTracks`, is a request for policy change, while the other three perform data access. Each of these operations is explained below.

`QT_OpenMovie` takes a string which represents a movie name and attempts to open it at every available fidelity level on the server. Each version is opened as a *track* of the base movie, and they are logically bundled as a single movie and returned. Along with a handle for the movie, akin to a file descriptor, `QT_OpenMovie` returns information about each track – specifically, the average bytes per second required to transmit each track across the network and the encoding method of each track. Upon opening, the best possible track is made the *active track*, and will remain active until the application requests otherwise. `QT_CloseMovie` frees up any resources associated with an open movie.

`QT_GetFrame` takes a movie handle, returned by `QT_OpenMovie`, and a time offset into the movie, and returns the first frame of the active track to be displayed after the offset. The frame is copied into the `ody_tsop` return buffer for use by the player. `GetFrame` also returns the index of the currently active track, so the application can properly decode the frame.

`QT_SwitchTracks` takes a movie handle, and a track identifier within that movie handle, and makes that track the new active track. Readahead is terminated for the old active track, and started for the new track. After the pre-read portion of the old track is exhausted, `QT_GetFrame` will return a frame from this new track. The new track will be used until another `QT_SwitchTracks` request is made.

6.2.2. The GRASS Warden

The GRASS warden provides two operations: `Grass Fetch`, which fetches a logical file from a server if not already cached, and `GrassSetQuality`, which determines which fidelity level future fetches will use. The final version of the system won't need `GrassFetch`: it'll have `open` redirected to it instead.

GRASS stores logical files in groups of related physical files. To avoid inconsistencies such as a raster header file showing the full size and a raster data file with lower resolution data, the prototype warden fetches files as a group. The GRASS warden currently makes no effort at cache replacement. Future refinements will address this as well.

6.3. The QuickTime Server and Player

The obvious fidelity dimension to exploit in video is the quality of individual frames; by reducing frame quality, we can also reduce bandwidth requirements. The QuickTime server currently stores movies at three different fidelity levels: full color uncompressed, full color with lossy JPEG compression, and black and white. Individual tracks can be opened, pre-read and closed. The server itself does not manage the different fidelity levels of the same logical movie as a unit; that is handled by the QuickTime warden.

The QuickTime player was modeled after a previously built standalone version that used the UNIX file system interface. It was redesigned to use the `ody_tsop` interface exported by the warden, rather than the standard UNIX file system interface. The new player opens a movie, finds the stream with the highest possible quality, and begins playing it. It also places a request to be notified if the bandwidth drops too low to support this track. If so, it switches to the new best possible stream. If, at some later time, bandwidth improves enough to allow playing a better track, the player will request a change.

Although the prototype explicitly trades performance for ease of implementation, the player has adequate performance in playing back movies, even at the highest quality. Of particular interest is the fact that the player was both simplified and functionally improved by the switch from the UNIX file system interface to that provided by Odyssey.

6.4. The GRASS Server and Applications

The server stores raster objects at three levels of fidelity, losing a factor of two in resolution for each degradation. Because the rasters are two dimensional, each degradation provides a savings of a factor of four in data size.

Applications wishing to open raster objects share a single routine in the GIS library. That routine first determines the estimated bandwidth available to the viceroy through the request interface with an empty bounds window, effectively polling the viceroy. Since no value could satisfy that bounds window, the bandwidth estimation is returned by the request call. The application then uses the `GrassSetQuality` operation to ask for a particular fidelity of raster. That fidelity is then cached on local disk for future use by GRASS applications.

7. Conclusion

Though rudimentary in many respects, our preliminary prototype has allowed us to gain initial validation of our ideas at low implementation cost. The results so far are encouraging. We have taken the source code of applications for two data types and have been able to restructure them into the Odyssey framework with modest effort.

We are now working toward a more complete and efficient prototype, motivated by two goals. First, we would like the prototype to support a broader collection of data types and associated applications. This will stress the designs of the Odyssey API and architecture, expose shortcomings, if any, and lead to refinements in both. It will also deepen our understanding of application-aware adaptation. Second, we would like the prototype to be better integrated with an operating system. An in-kernel implementation will allow more serious resource management, provide better performance and functionality, and enable more rigorous evaluation of our design.

As was discussed early in this paper, the constraints of mobile computing lead inevitably to the recognition that adaptivity is essential in any system that provides mobile data access. But although the general importance of adaptivity has been recognized by many researchers[2, 3, 6, 12, 13], we are not aware of specific system designs, much less implementations, that support application aware adaptation. The work reported here thus represents a journey into uncharted waters.

References

[1] APPLE COMPUTER, INC. *Inside Macintosh: QuickTime*. Addison-Wesley Publishing Company, 1993.

[2] DUCHAMP, D. Issues in Wireless Mobile Computing. In *Proceedings of the Third Workshop on Workstation Operating Systems* (Key Biscayne, FL, April 1992).

[3] FORMAN, G. H., AND ZAHORJAN, J. The Challenges of Mobile Computing. *IEEE Computer 27*, 4 (April 1994).

[4] HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems 6*, 1 (February 1988).

[5] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems 10*, 1 (February 1992).

[6] KULKARNI, D. C., BANERJI, A., CASEY, M. R., AND COHN, D. L. Information Access in Mobile Computing Environments. Tech. Rep. TR-93-11, University of Notre Dame, Notre Dame, 1993.

[7] MADRY, S. Geographical Resources Analysis Support System (GRASS), an Integrated Public Domain GIS and Image Processing System. In *GIS/LIS 1989 Proceedings* (Orlando, FL, November 1989).

[8] SALTZER, J., REED, D., AND CLARK, D. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems 2*, 4 (November 1984).

[9] SATYANARAYANAN, M., KISTLER, J. J., KUMAR, P., OKASAKI, M. E., SIEGEL, E. H., AND STEERE, D. C. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers 39*, 4 (April 1990).

[10] SATYANARAYANAN, M., NOBLE, B., KUMAR, P., AND PRICE, M. Application-Aware Adaptation for Mobile Computing. *Operating Systems Review 29*, 1 (January 1995). Also available as Tech. Rep. CMU-CS-94-183, Carnegie Mellon University, School of Computer Science.

[11] SIDEBOTHAM, R. Volumes: The Andrew File System Data Structuring Primitive. In *European Unix User Group Conference Proceedings* (August 1986). Also available as Tech. Rep. CMU-ITC-053, Carnegie Mellon University, Information Technology Center.

[12] THEIMER, M., DEMERS, A., AND WELCH, B. Operating System Issues for PDAs. In *Proceedings of the Fourth Workshop on Workstation Operating Systems* (October 1993), IEEE.

[13] WEISER, M. Some Computer Science Issues in Ubiquitous Computing. *Communictions of the ACM 36*, 7 (July 1993), 75–84.