Experience with adaptive mobile applications in Odyssey

B.D. Noble [a] and M. Satyanarayanan [b]

[a] *Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 40109, USA*
[b] *School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA*

In this paper, we present our experience with *application-aware adaptation* in the context of *Odyssey*, a platform for mobile data access. We describe three applications that we have modified to run on Odyssey – a video player, a Web browser, and a speech recognition system. Our experience indicates that it is relatively simple to incorporate applications into Odyssey, and that application source code is not always essential. Although our applications were built without knowledge of each other, Odyssey is able to run them concurrently without interference. However, our experience also exposes important areas of future work. Specifically, it reveals the difficulty of balancing agility with stability in adaptation, and emphasizes the need for controlled exposure of internal Odyssey state to users.

1. Introduction

The resources available to a mobile client vary widely and unpredictably, requiring the client to adapt its behavior to these changes. In the face of diverse, concurrent applications such adaptation is best provided through a collaboration between these applications and the operating system. This collaborative approach is called *application-aware adaptation*. Odyssey, a platform for mobile data access, is the vehicle we have built to explore application-aware adaptation.

Earlier papers have described the programming interface Odyssey provides for adaptive applications [15], and have presented results from controlled experiments to validate the benefits of application-aware adaptation [16]. In this paper we report on the qualitative lessons we have learned in building Odyssey applications, and refining them in the light of usage experience. These lessons can be summarized as follows:

- It was relatively easy to modify our example applications to take advantage of Odyssey.
- An important class of binary-only applications can be incorporated into Odyssey without source-level changes.
- Odyssey effectively supports competing applications without requiring them to have explicit knowledge of one another.
- Striking the right balance between agility and stability is difficult; care is required to ensure that the adaptation policy provides a satisfactory experience for the user.
- Users benefit from some form of feedback about adaptations as they occur. Likewise, their experience is improved by feedback concerning the causes of those adaptations.

The paper begins by presenting a brief overview of Odyssey, and then describes the three applications we have built on it. For each application, we first describe its integration with the Odyssey prototype; we then describe the adaptation policy used for that application. We conclude by reporting on the lessons we have learned so far from our experience with these applications.

2. Overview of Odyssey

2.1. Data fidelity

When faced with a sudden scarcity of resources, a mobile client can react in two ways. First, it can reduce its demand for a scarce resource by substituting a more plentiful one. Second, it can reduce the quality of data it is accessing, thereby reducing resource consumption.

There are many ways in which plentiful resources might be traded for scarce ones. For example, lossless compression applied to transmitted data trades computation for bandwidth. Caching to avoid future data exchanges trades disk space for both bandwidth and computation. Rather than using a remote computation engine that is difficult to reach, the client may perform some complex action locally. A mobile client with a software-controlled radio may increase power to decrease bit-error rate, or reduce power consumption by reducing bandwidth [3].

However, the trading of resources often is not sufficient to cope with large swings in availability. As an example, the bandwidth available to a mobile client can vary by several orders of magnitude. Further, the scarcity of resources available to a mobile client limits the degree to which such trades can be made. These together require a more aggressive form of adaptation. Odyssey defines adaptation as the trading of data quality for resource consumption. For example, when the bandwidth available to a video player drops, it could switch to a video stream with fewer colors and coarser resolution rather than suffer dropped frames. Likewise, when the bandwidth available to a Web browser drops, it might fetch images that have been aggressively compressed rather than wait an inordinate amount of time for the full quality versions.

Odyssey defines a new property, *fidelity*, to quantify this notion of quality. For any data item, there exists a most complete, current, detailed version of that item called the *reference copy*. Ideally, the reference copy is always presented to the mobile user. However, when resources become scarce, the item will be degraded in some way. Fidelity is defined as the degree to which a presented item matches the reference copy.

Fidelity is necessarily a type-specific notion; different kinds of data are degraded in different ways. For example, video may be degraded by dropping frames, reducing image size, or reducing image quality of individual frames. Maps can be degraded by either coarsening the level of detail, or removing certain classes of features; for example, one might only look at roads and rivers, ignoring buildings.

This notion of fidelity requires data to be relatively rich in structure; an untyped stream of data cannot be degraded in any meaningful way beyond relaxing consistency constraints. Therefore, Odyssey's target applications are those which access relatively rich, and hence resource-demanding, data.

Note that the trading of fidelity for resource consumption is different from what is often meant by adaptation. For example, TCP adapts its window size in an effort to maximize throughput without causing undue congestion. This sense of adaptation is necessary on a mobile node, but is not sufficient. When bandwidth drops by several orders of magnitude, a drastically reduced window size will allow the mobile client to get the best throughput possible. However, data will then arrive at the client at too slow a rate to be acceptable.

2.2. Model of adaptation

In Odyssey's view, the operating system is responsible for monitoring resource availability, notifying applications of relevant changes to those resources, and enforcing resource allocation decisions. In contrast, each application is responsible for deciding how best to exploit available resources. This approach best supports adaptation by concurrent and diverse applications; it is called *application-aware adaptation*.

The merits of application-aware adaptation can be best understood by considering its alternatives. Since fidelity is a type-specific notion, one might be tempted to place responsibility for adaptation entirely with the application. We call this approach *laissez-faire adaptation*, and it is exemplified by work such as RLM [11] and Cen's feedback-adaptive toolkit [2,7]. Such approaches do allow individual applications to adapt precisely according to their own goals, and do not require any operating system support. However, concurrent applications that use *laissez-faire* adaptive schemes are unlikely to behave well. They each adapt to the same set of environmental changes, and compete for the same set of scarce resources. Without some central authority, they are likely to interfere with one another and adapt at cross purposes; *laissez-faire* adaptation cannot support concurrency.

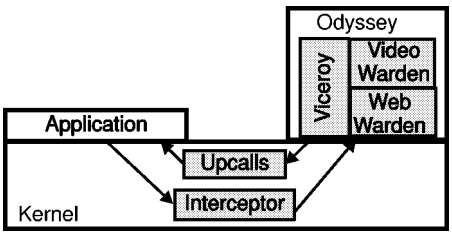

Figure 1. Odyssey architecture.

One could remedy this shortcoming by making the system entirely responsible for adaptation, an approach called *application-transparent adaptation*. Examples of such systems include Coda [14] and Bayou [21]. Legacy applications can continue to run on these systems unmodified, as they are not involved in adaptation decisions. However, they cannot support different applications that, when using the same data in the same circumstances, desire different adaptive behavior; application-transparent adaptation cannot support diversity.

2.3. Client architecture

The architecture of an Odyssey client is shown in figure 1. The bulk of the architectural components reside in user space for simplicity of implementation. However, they should be thought of as part of the operating system, and could be implemented either directly in the kernel or as a middleware layer between the kernel and the applications.

Odyssey objects are available to applications as part of the file name space. The *interceptor* provides a VFS client [9] that forwards file system requests on Odyssey objects to the *viceroy*; this latter component is responsible for all type-independent functionality on the client. The viceroy's most important task is monitoring the availability of resources and managing their use. To do so effectively, the viceroy must have knowledge of all resource usage on behalf of applications in the system.

Since fidelity is a type-specific notion, each Odyssey object must carry with it a notion of type. The *wardens* are a set of type-aware code components, one warden per type. Wardens are chiefly responsible for providing a menu of fidelities from which applications can pick. Wardens can also provide semantically rich access methods to applications, and can take advantage of type knowledge to optimize resource usage, consistency, and so on.

2.4. Programming interface

Odyssey adds two calls to the standard system API. The first, *resource request*, is used to inform Odyssey of the changes to resource availability of interest to the calling application. For example, a video player might wish to know if bandwidth ever drops below a certain level. The second, *type-specific operation*, is used by an application to change the fidelity at which data is accessed. In the

request(**in** path, **in** resource-descriptor, **out** request-id)

(a) Placing a Request

resource id
lower bound
upper bound
name of upcall handler

(b) Resource Descriptor Fields

Figure 2. Resource requests.

example above, the video application might reduce frame rate when bandwidth does drop significantly.

The request API takes advantage of the fact that not all changes in resource availability need be made known to an application; only certain changes are of interest. For example, consider a video player showing a particular movie at a medium level of fidelity. Small changes in bandwidth are unimportant, but large enough changes may warrant raising or lowering fidelity. The range defined by these limits is called a *window of tolerance*.

When an application chooses a fidelity level, it tells the system of any relevant tolerance windows by issuing a resource request, shown in figure 2. A request names some particular resource, the window of tolerance on that resource, and a function to call when the resource strays outside of the specified window. Resources of interest include bandwidth, latency, disk space, available compute power, remaining battery power, and money.[1]

These resource requests are forwarded to the viceroy, which records them. As it updates its estimates of resource availability, it checks these estimates against any established windows of tolerance. If a resource strays outside of one or more requested windows, each affected application is notified via an *upcall*. This mechanism allows the viceroy to filter out uninteresting changes in availability, and focus on those of importance to some application.

Upcalls are treated much like signals, but they are delivered with exactly-once semantics rather than at-most-once, and may pass arguments and return values. A resource notification upcall carries with it the resource whose value has changed along with the new value.

The application responds to this notification by changing the fidelity of the data it is accessing; these fidelity changes are carried out by wardens. Placing the responsibility with the warden amortizes the effort required to support fidelity changes across all applications using that warden's type.

There may be many different wardens, each with many different fidelity-changing operations. Rather than attempt to encode all possible such operations into the API, they are all multiplexed over a single new system call: the type-specific operation, or *tsop*. Figure 3 summarizes the tsop call, which is similar to the POSIX system call `ioctl`.

[1] Money was included in anticipation of the use of electronic payments for fee-for-service components of the mobile infrastructure.

tsop(**in** path, **in** opcode, **in** insize, **in** inbuf, **inout** outsize, **out** outbuf)

Figure 3. Type-specific operations.

It passes an unstructured argument buffer to the warden, and receives the result of the operation, if any, in another unstructured buffer.

In addition to providing fidelity-changing operations, the tsop mechanism can provide type-specific access methods to applications. Instead of a limited, byte-oriented open, close, read, and write interface, the warden can export an interface more semantically meaningful for the type in question. For example, the video warden exports an interface allowing applications to read a movie as frames of video, rather than simply bytes in a file.

2.5. Managing resources: Bandwidth

The principal resource of interest in our current prototype is network bandwidth. We chose to start with bandwidth because, for mobile nodes, it is the most volatile of the resources; it can vary over several orders of magnitude without warning.

The bandwidth estimation algorithm makes use of observations at the transport layer, a user-level implementation of an adaptive window-size, reliable transport protocol called RPC2 [19]. Each exchange between client and server is timed and logged, and the viceroy applies a simple linear filter to smooth these observations. The result is a prediction of the near-term total bandwidth available to the machine.

This total bandwidth must be divided amongst the applications competing for it. The viceroy divides most of it based on recent use; those applications that have consumed a larger share in the immediate past are assumed to need a larger share in the immediate future. However, a small portion is reserved and divided fairly; this is to avoid unduly punishing applications that do not use the network for extended periods of time.

2.6. Programming model

The programming model underlying this API is an *adaptive decision loop*. Each Odyssey application implements its decision loop outside the main control flow of the application. The loop is invoked by a resource notification upcall. On notification, the loop first selects a fidelity appropriate to the new resource situation. It then asks for that new fidelity via a tsop, and sets any required windows of tolerance appropriate to the new fidelity. When finished, it awaits the next notification. This isolation of control preserves simplicity when adding adaptive capabilities to an application.

The implication of this structure is that the main body of an Odyssey application must cope with data at varying levels of fidelity. Fortunately, this is already true for a

large class of interesting applications. For example, most video players can already decode a number of different video representations. Further, most complex data types are self-descriptive. For example, a JPEG [23] image is decoded exactly the same way, regardless of the quality of compression or the number of colors in the color map. A map can be rendered by the same code, whether or not the roads are present. As a result of the positive experience reported in this paper, we now have greater confidence in this programming model.

3. Example applications

To gain first-hand experience with application-aware adaptation, we modified a small set of applications to run on Odyssey. We chose these applications with an eye toward answering the following questions:

- How much effort is required to modify applications to take advantage of Odyssey?
- Can Odyssey effectively support a number of concurrent, diverse applications using different data types?
- Is source code essential? Can *shrink-wrapped* applications – those for which no source is available – also take advantage of Odyssey?

The three applications are: XAnim, a video player; Netscape, a web browser for which source was not available; and Janus, a speech recognition system. Each application uses a different data type, and has its own warden and server, giving a broad range of experience. They are all real applications that enjoy significant use, and each presents unique challenges in integration with Odyssey.

Data can be presented at a number of discrete fidelities, each of which is assigned a fidelity metric between zero and one; a fidelity of one corresponds to the reference copy of the data item. Fidelity levels provide a total ordering on quality of data, but do not necessarily provide a measure of relative quality perceived by the end user. Each application also has a performance metric. The central addition to each application was some *adaptive policy* – the strategy that application would use in trading fidelity for performance. We had no prior experience in designing adaptative policies, and so made choices for each application that were reasonable and stressed the Odyssey API.

Sections 3.1–3.3 describe the three applications. Each section describes the application, warden, and server, as well as how these components are integrated into the Odyssey prototype. Each section provides an estimated count of modified or new lines of C code for each of the applications. It then defines the fidelity and performance metrics, and the fidelity policy of the application.

3.1. Video player: XAnim

The first application added to Odyssey was *XAnim*, a video player whose source code is publicly available [17]. In its original form, XAnim reads a movie file from a local disk and plays it back to the screen, skipping late frames to maintain pace through the file; it was approximately 57 KLOC (thousand lines of code). The main data type used by this player in the context of Odyssey is QuickTime, a standard video format defined by Apple Computer [1]; this format has an explicit time base in which the video stream is encoded, and provides facilities for many different representations.

3.1.1. Integration with Odyssey

The monolithic XAnim application was split into a client, warden, and server. Adding these components to Odyssey was straightforward; Figure 4 illustrates the resulting structure. The server is a relatively simple piece of code consisting of five KLOC, of which one KLOC were added to or modified from the original code base. It stores each movie as a number of pre-computed versions called *tracks*, each providing a different fidelity of the movie. Storing these additional fidelities avoids computational overhead when delivering degraded video, at a space overhead cost of 60%. Of course, one could easily modify this server to use an on-line scheme that degraded video images on the fly [6].

The number and sizes of tracks available for each movie are part of that movie's meta-data. The meta-data also specifies, for each track, the sizes and offsets of each frame in that track. The warden, which handles QuickTime data at the system level, can obtain this meta-data from the server, and fetch a range of bytes from a particular track. The warden is responsible for mapping a track's frame numbers to byte ranges within that track; the server provides no such mapping. The warden also prefetches data from the server, anticipating that the most common client behavior is sequential access within a single track. Like the server, the warden is a relatively small piece of code at 2.5 KLOC.

The client obtains movie data entirely through type-specific access methods, rather than the more cumbersome file I/O interface. This enables a simplification of the client, removing 7.5 KLOC from the original code base. The type-specific operations supported by the QuickTime warden are summarized in figure 5, and described below.

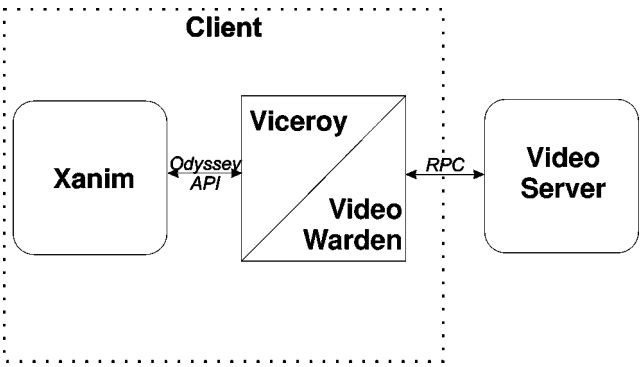

Figure 4. Integration of XAnim with Odyssey.

`qt_open`	Open a QuickTime movie, return a vector of track descriptions, one per track.
`qt_getframe`	Get frame number N from track T
`qt_getframe_at`	Get the first frame to be displayed at or after time t from track T.

Figure 5. Access methods of QuickTime warden.

On `qt_open`, the warden fetches the movie meta-data and builds two frame maps. The first maps time offsets to frame numbers for each track of the movie. The second maps frame numbers to byte ranges. Once these are constructed, the warden returns the track summary to the application that called `qt_open`.

On `qt_getframe`, the warden translates the frame number to a byte range. If the requested frame is in the warden's prefetch buffer, it is returned. If it is not present, the warden returns the frame from a better quality track if it is present. If neither of these are present, the warden fetches it from the server on demand. Less than one KLOC needed to be added to the client to use this interface, in place of the 7.5 KLOC removed from the original.

A background thread belonging to the warden prefetches frames from the last track requested by `qt_getframe`; this prefetching is controlled by two parameters, `highwater` and `lowwater`. Whenever there are fewer than `lowwater` bytes in the prefetch buffer, the warden prefetches frames until the prefetch buffer contains at least `highwater` bytes. If higher-quality frames are present when an application requests a lower-fidelity track, the better frames are retained. Conversely when bandwidth improves, lower-quality frames are discarded and the requested higher-quality ones are fetched on demand.

3.1.2. Metrics and policy

In the prototype implementation, each movie consists of three tracks. These three versions are: JPEG-compressed color at quality 99, JPEG-compressed color at quality 50, and black-and-white. There is no inter-frame compression. Each track is encoded at ten frames per second.

Individual frame fidelities are assigned as 1.0, 0.50, and 0.01 to JPEG(99), JPEG(50), and black-and-white frames, respectively. Over a single execution of the player, the single achieved fidelity metric is the average of the fidelities of the displayed frames; higher average fidelity means that displayed frames were, on average, of a better quality. Thus a movie with half of its frames displayed from each of the two best tracks would have a fidelity of 0.75.

The client's performance metric is the number of frames it is forced to skip due either to late-arriving frames or a delay in the decoding of some previous frame. If a frame arrives after its deadline, it will be dropped rather than shown. If a frame is more than one frame-time late, then the client will skip past frames that should have been shown while the late frame was being obtained. Of course, user perception of playback quality is influenced by many factors. For instance, some number of frames dropped consecutively will be perceived as worse than the same number dropped intermittently. This is an important issue that we are still grappling with.

The client's adaptation policy is to play the best quality track possible without dropping any frames. When the client opens a movie with `qt_open`, it calculates the bandwidth required to play each track in the movie. From these calculations, the client derives a set of bandwidth ranges appropriate to each fidelity. These ranges are defined with some overlap, and select for fidelities aggressively. The lower bound in a track's range is set to 95% of the bandwidth nominally required to support it; the upper bound is the minimum nominally required for the next higher track.

After opening a movie, the client places a resource request on the bandwidth for the highest quality track, and begins playing frames from that track. Whenever it is notified that the bandwidth has strayed outside of the bounds for the current track, it changes the track from which it is requesting frames, and places a resource request appropriate to the new track's bandwidth requirements.

This encoding of QuickTime data is well-suited to an adaptive policy that switches between tracks. This is because each frame can be rendered in isolation, without need for some other reference frame. We have begun to remove this restriction for formats that use inter-frame compression, such as MPEG [8], by only switching tracks at reference frames. This keeps the track switch point relatively seamless from the user's point of view. Since the definition of a reference frame is type-specific, enforcing such constraints is properly the responsibility of the warden.

3.2. Web browser: Netscape

Our second application was *Netscape*. At the time we began this work, source code for Netscape was not publicly available. It was chosen expressly as an example of a shrink-wrapped application that can take advantage of Odyssey's support for adaptation.

3.2.1. Integration with Odyssey

To cope with the lack of source code, Odyssey makes use of Netscape's *proxy* facility. Netscape can use this facility to route all of its HTTP [5] requests for data through a designated process. This process is commonly on a remote host; such a remote process might act as a gateway that is exempt from firewall restrictions, or a caching proxy for a group of machines [10]. In our case, we place the proxy, called the *cellophane*, on the client between Netscape and Odyssey. The cellophane redirects Netscape's requests through the file system to Odyssey. From the point of view of Odyssey, the cellophane is the adaptive application; it is quite small, at three KLOC.

The Web is integrated into the Odyssey name space as a single object. Lookup operations on that object will at-

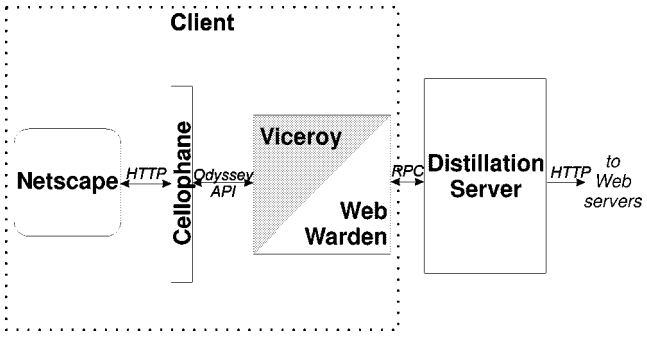

Figure 6. Integration of Netscape with Odyssey.

`webw_setqual`	Set desired quality for images
`webw_getqual`	Obtain desired quality for images
`webw_sethdr`	Pass along request headers for the next requests
`webw_stat`	Obtain the headers for a particular page

Figure 7. Type-specific operations of Web warden.

tempt to resolve the name component as a URL in the Web. Since URLs use the '`/`' character, which is also used as the UNIX component separator, the cellophane must convert all instances of '`/`' appearing in a URL to backslashes.

The Web warden, which is less than five KLOC, forwards all of the cellophane's requests to a remote *distillation server*, which is presumed to be well-connected to the rest of the Web. The distillation server, at five KLOC, fetches HTML pages and associated images in response to requests, and forwards them to the HTML warden. It is capable of degrading images on the fly using JPEG compression to shorten their transmission time to the client. These components – cellophane, warden, and distillation server – are depicted in figure 6.

The Web distiller focuses on images for two reasons. First, they make up a majority of all bytes served on the Web [13]. Second, there exists a natural degradation method – JPEG compression – that gives good size reductions while yielding tolerable quality. No such obvious degradation exists for text in HTML pages.

The Web warden exports four type-specific operations; two provide mechanisms to change the fidelity of images, and two are used to help integrate the Web into the client's file system. These tsops are summarized in figure 7.

The `webw_setqual` operation is used by the cellophane to set the desired level of compression for images fetched. It is set based on current bandwidth as described in section 3.2.2. The `webw_getqual` operation is used by the cellophane to query the current quality setting.

The other two operations handle meta-data for HTTP requests and responses. A request for a Web object carries with it *request headers*, which can affect the content of the fetched objects. To pass these request headers to the Web warden, an application uses the `webw_sethdr` operation. Each page can have associated with it a separate set of meta-data, which an application can obtain via the `webw_stat` call.

3.2.2. Metrics and policy

The distillation server, in addition to passing images unchanged, has three distinct levels of degradation, for a total of four levels of fidelity. These levels of degradation consist of JPEG compression at quality levels 50, 25, or 5; lower numbers produce lower-quality images with smaller representations. These degraded qualities are assigned fidelity levels of 0.5, 0.25, and 0.05, respectively; the original image is assigned a fidelity level of 1.0. The distillation server degrades only those images for which it is expected to provide a benefit – images 2 KBytes or larger. For smaller images, the effort to distill them takes longer than simply forwarding them on all but the very slowest of networks.

The performance metric for Netscape is the time to load and display a particular HTML object. Netscape's adaptation policy is to load the best quality image possible within twice the expected time to load the reference quality image at 10 Mb/s. This heuristic is based on the following intuition: there is little utility in loading an image faster, since users typically are willing to wait roughly as long as it might take over an Ethernet. However, much longer waits, albeit for better quality images, are not likely to be tolerated. This policy was our initial, educated guess about what sort of delay users would find tolerable; we have not yet carried out the human factors experiments to validate this choice.

For each of the four fidelity levels available, the cellophane selects a bandwidth range appropriate to that fidelity level. These ranges are currently hard-coded in the cellophane, and were based on a small set of experiments measuring times to perform JPEG compression and resulting reduction in size. As the bandwidth between the warden and distillation server changes, the cellophane adjusts the distillation level of images served.

3.3. Speech recognizer: Janus

The final application modified to take advantage of Odyssey is *Janus* [22], a speech recognition system. Janus takes as input a raw, sampled speech utterance collected from a microphone, and returns the ASCII representation of the utterance. This process is very expensive in both CPU cycles and virtual memory. Since a mobile client is relatively under-powered, it would be useful to offload this computation whenever possible.

The recognition process has two phases. The first is *vector quantization* [18], a signal processing step that transforms the raw speech utterance into a much more compact representation. This phase is relatively inexpensive to compute. The second phase consists of the remainder of recognition, and comprises the bulk of the processing required in recognition.

This application is quite different from the others; speech data is not something that is merely accessed, but rather it

is generated at the client and then transformed, possibly elsewhere. Further, speech recognition presents considerable potential as well as challenge for mobile systems. It is especially useful when mobile since it leaves the user's eyes and hands free for other activities [20]. However, the resource requirements for high-accuracy speech recognition are substantial, especially when mobile, since background noise is often high. Adding higher-level semantic processing, such as language translation, leads to even greater demands on computing resources.

3.3.1. Integration with Odyssey

There is a single, distinguished speech object in the Odyssey namespace. Writing an utterance to this object begins recognition; a subsequent read on that object will return the recognized text. A simple front end, consisting of under one KLOC, runs a loop that collects the raw speech utterance, writes it to the speech object, and reads the result.

Each utterance is forwarded to the speech warden, which is approximately two KLOC. The warden can use two different recognition servers, each a full copy of Janus with one additional KLOC to handle communication. These components are illustrated in figure 8.

The warden, when passed a speech utterance, can recognize it in one of three ways. First, it can pass the raw, large utterance to the remote Janus server for full recognition; this is called *remote* recognition. Second, it can pass the utterance to the local Janus server for *local* recognition. Third, it can pass the utterance to the local server for just the vector quantization step, and pass the much smaller result to the remote server for the second phase of recognition; this is called *hybrid* recognition. The warden provides a single type-specific operation, `speech_setstrat`; the front end can use this to select one of the three strategies.

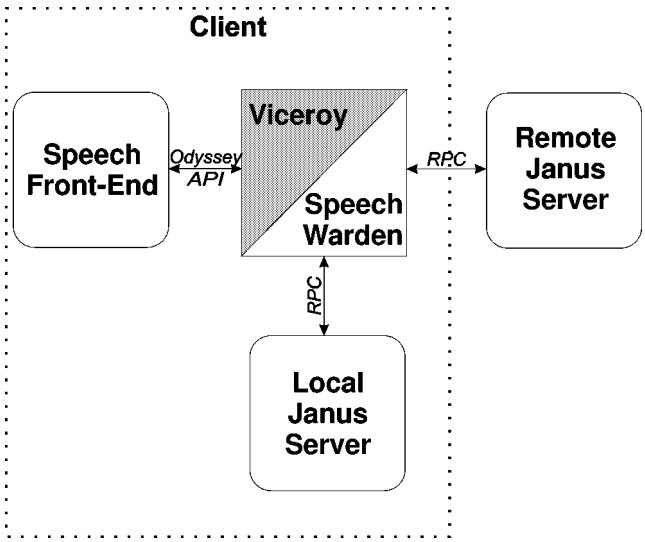

Figure 8. Integration of Janus with Odyssey.

3.3.2. Metrics and policy

The fidelity metric is very simple in Janus. When performing remote or hybrid recognition, the recognition is the best that Janus can possibly do given a particular vocabulary and grammar. When performing entirely local recognition, the mobile node uses a smaller acoustical model, grammar, and vocabulary. This is an attempt to reduce the computational requirements of speech recognition, so that other tasks may run concurrently; full Janus recognition would swamp all other tasks.

The full recognition is assigned a fidelity of 1.0, the lower quality one a fidelity of 0.5. To help the user cope with this changing accuracy, Janus provides feedback about its behavior via a synthesized voice. This voice informs the user of increases or decreases in the accuracy of recognition.

The performance metric is latency: the time it takes to recognize a speech utterance. The front end knows how fast the server is relative to the client, and can estimate the time it will take to recognize an utterance of a particular length completely on the client, completely on the server, or in hybrid recognition. In the current prototype, we determined these costs empirically and coded them into the decision loop. We are working to incorporate computational power as a resource so that Janus can correctly function on any combination of hardware platforms.

When the front end is given an utterance to recognize, it decides whether remote or hybrid recognition will result in the fastest response; this decision is based on the current bandwidth estimation. If network bandwidth is sufficiently high, remote recognition is best; otherwise, hybrid recognition is best. Because of the severe computational demands of the second phase of recognition, and its lower quality, the front end does not use local recognition unless the connection to the remote host is down, leaving it no alternative.

4. Lessons learned

These applications have been studied as customers of Odyssey's programming interface, and have been under constant improvement for the past year. In the course of building and experimenting with these applications, we validated many of our assumptions about adaptive applications. We have also learned some important lessons along the way. It is important to point out that these lessons are drawn from a fairly small set of example applications, and that we are actively adding to our understanding of them. We plan to increase the number of applications and data types within Odyssey, and to gain experience in their use.

4.1. Ease of porting

We began this work in the belief that it would not be difficult to modify programs to make use of application-aware adaptation, for two complementary reasons. First, the programming model places the adaptive code outside the main

body of the application. This both limits the scope of necessary modification, and reduces the degree to which the programmer must understand the application in question.

The second reason is that the applications we had in mind were already capable of decoding multiple representations of a given data type. Therefore, the decision loop can change fidelity and rely on the application to handle it correctly. Some care must be taken to avoid violating application invariants when doing so. For example, our video warden does not change fidelities until forwarding a reference frame – one that can be decoded without reference to other frames.

In fact, it was not unduly difficult to modify our sample applications to take advantage of application-aware adaptation. Each one required at most a few thousand lines of C code. Further, type-specific access methods such as the video warden's frame-level interface allow substantial simplification of an application by removing the code required to map from the file I/O model to more abstract constructs.

4.2. Shrink-wrapped applications

There are many applications for which source is not available but are nevertheless extremely important. This is true on UNIX-like platforms as well as the Windows variants. For example, Adobe's *Acrobat Reader* is distributed as a binary-only program, and is the standard display application for an indispensable document format [12].

As of this writing, source code to the Netscape browser is publicly available. However, when we began this work, it too was shrink-wrapped. Despite this, incorporating Netscape into Odyssey was straightforward. Netscape's proxy mechanism provides a convenient point at which all data access operations can be intercepted. The intercepting agent – in our case, the cellophane – plays the role of the adaptive application from the operating system's point of view, and the source of data from the application's point of view. One could interpose such an agent between any proxy-capable application and the outside world.

Obviously, shrink-wrapped applications without a proxy mechanism would require a more sophisticated approach for adaptation. However, we believe that such applications can still be incorporated into Odyssey. For example, one could link such applications against a run-time library that intercepted all file and network system calls. The library could then re-route these system calls to a cellophane-like component.

The provision of fidelity levels is the responsibility of the warden. Therefore, the data used by these applications must be in a known format. Without this important restriction, changes in fidelity could not be provided in any sensible way.

4.3. Burden on applications

Our model of collaboration between applications and the operating system provides for a strict division of duties. The operating system manages resource usage across the machine, while each application need only consider its own goals in making adaptation decisions. Whether or not applications could remain unaware of competitors was an open question when the work began. For instance, it is possible that concurrent applications would land in unstable adaptation ranges, constantly switching between fidelities.

Our experience to date suggests that applications do not need to be aware of one another in the Odyssey framework. While unstable ranges are possible, the applications do not enter them in practice. This is true independent of relative load of the applications. However, we suspect that collaborative applications that choose to make their relationship known to the system can make even better adaptation decisions. For instance, in a collaborative task that included audio and video conferencing and a shared whiteboard, the viceroy could be told that video bandwidth should be sacrificed in favor of the audio stream or the whiteboard updates.

4.4. Balancing agility and stability

While the adaptive policies embodied in our sample applications are reasonable first attempts, some care is required to produce effective adaptive algorithms. As an anecdotal example, consider our current version of XAnim. The perceived difference in quality – the *perceptual distance* – between the highest and middle video fidelities is not great. One must look closely to detect when a change in fidelity between these two levels has occurred. As a result, it is not very disconcerting to a user if the video player switches rapidly between them. However, the perceptual distance between the middle fidelity and the lowest one – between low color and black and white – is very large. Each time the video player switches between these fidelities, it is immediately noticeable to the user. The experience of watching the video player rapidly switch between these two fidelities is jarring.

To account for this, we are exploring the idea of using perceptual distance to limit the rate at which changes are tolerated by the video application. When bandwidth drops substantially, the video player must switch to the black-and-white fidelity, else it will lose frames. However, the player is sceptical of a subsequent increase in bandwidth. Rather than aggressively switching back to the medium or high fidelity, the player waits a few seconds. If the increase is transient, the player will not be forced to oscillate. If the increase is long-lived, the player will show sub-optimal frames until switching, but the overall user experience is improved.

Two things are required to quantify and ground this technique. First, we must better understand how to assign fidelity metrics to data. In addition to having total ordering, fidelity should express relative quality in terms of user perception. Second, we need to be able to map from quantitatively meaningful fidelities to a good adaptive algorithm. Such an algorithm would scale reluctance to improve from one fidelity to another with the difference between those fidelities.

4.5. User feedback

In a world as dynamic as that of a mobile client, users require some feedback about when a change has occurred, and why. For the video application, changes that affect quality are evident; the movie provides a time base that provides an obvious way to track fidelity changes. For Netscape, the feedback is more subtle. It is not clear that changes in quality from one page to the next can be tracked by users; they may blame the page author rather than their network connection for a poor quality image.

We anticipated the need for some amount of feedback to users. For example, when Janus switches to lower fidelity recognition, the user is informed by a synthesized voice. Likewise, when connection to the remote server is reestablished, the user is also informed. Without such a mechanism, the only feedback available would be less accurate recognition; it might take several utterances for the user to pinpoint what has gone wrong.

This idea of adding mechanism to provide feedback could be extended to revealing information about the state of the mobile client generally; users of our demonstration system sometimes report confusion about why fidelity changes happen when they do. For example, a small light in a corner of the screen could indicate network health; a green light would indicate a strong network connection. A weak connection would be signified by a yellow light, and a red light would indicate that the client is completely disconnected. Having such a mechanism could help the users explain fidelity changes, reducing the potential for frustration while using the system. Ebling [4] has demonstrated the importance of such a *translucent* interface for usability in mobile computing.

Such feedback can go both ways. There are cases where the user can help Odyssey make better decisions about how to divide up available resources, since not all applications are of equal importance to him. For example, the user might be intently watching a video report from a news agency, while only paying cursory attention to financial reports in the background. If the user's preferences could be made known to the system in a simple way, the news report could be allocated more bandwidth.

4.6. Writing adaptive applications

Our experience also suggests some guidelines for those writing new adaptive applications. The first rule of thumb is to craft the application in such a way that decoding and using various representations is quite natural. This renders the division of control and processing simple. Second, one must have a reasonable picture of resource consumption at various levels of fidelity, as well as how those fidelity levels map to user experience. Third, the placement of type-specific access methods in the warden can greatly simplify each application using that type; doing so is worthwhile.

5. Conclusion

Application-aware adaptation shows significant promise as a technique to cope with the constantly changing world of a mobile client. We have constructed a prototype system, called Odyssey, that provides adaptive services to applications, and modified three real-world applications – a video player, a Web browser, and a speech recognition system – to take advantage of those services.

In the course of building and experimenting with these applications, we have learned several things about adaptive applications generally. They are not particularly difficult to construct, even in one case where source code to the original application was not available. They can be built in isolation, but run together without undue difficulty. While there is not yet a principled technique to construct adaptive algorithms that provide the best user experience possible, there is clearly a need to provide simple feedback to the user about what the system is doing. Our work in this area is ongoing; we are adding more applications and data types to Odyssey. As we gain experience in using them, we hope to better understand what makes an adaptive algorithm better or worse from a user's point of view.

Acknowledgements

The development of Odyssey and its applications is the product of the effort of many people; without them, this work would have been impossible. Jason Flinn modified Janus to produce the speech server, warden, and client. Likewise, Dushyanth Narayanan incorporated XAnim into the Odyssey framework. Eric Tilton produced the infrastructure supporting Netscape, and Kevin Walker built much of the measurement infrastructure that led to the lessons reported here.

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA), Air Force Materiel Command, USAF under agreement number F19628-96-C-0061, the Intel Corporation, AT&T, and Lucent Technologies.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Intel, AT&T, Lucent, DARPA, or the U.S. Government.

References

[1] Apple Computer, Inc., *Inside Macintosh: QuickTime* (Addison–Wesley, 1993).

[2] S. Cen, A software feedback toolkit and its application in adaptive multimedia systems, Ph.D. thesis, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology (October 1997).

[3] S. Chen, N. Bambos and G. Pottie, On power control with active link quality protection in wireless communications networks, in: *IEEE Proc. 28th Annual Conf. on Information Sciences and Systems* (1994).

[4] M.R. Ebling, Translucent cache management for mobile computing, Ph.D. thesis, School of Computer Science, Carnegie Mellon University (March 1998). Also appears as technical report No. CMU-CS-98-116.

[5] R. Fielding, J. Gettys, J. Mogul, H. Frystyk and T. Berners-Lee, Hypertext transfer protocol – HTTP/1.1, Internet RFC 2068 (January 1997).

[6] A. Fox, S.D. Gribble, E.A. Brewer and E. Amir, Adapting to network and client variability via on-demand dynamic distillation, in: *Proc. of the 7th Int. ACM Conf. on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA (October 1996).

[7] J. Inouye, S. Cen, C. Pu and J. Walpole, System support for mobile multimedia applications, in: *Proc. of the 7th Int. Workshop on Network and Operating Systems Support for Digital Audio and Video*, St. Louis, MO (May 1997).

[8] ISO/IEC JTC1/SC29/WG11, Coding of Moving Pictures and Associated Audio for Digital Storage Media up to 1.5 Mbit/s. MPEG, Int. Standard, ISO 11172.

[9] S.R. Kleiman, Vnodes: An architecture for multiple file system types in Sun UNIX, in: *Summer Usenix Conf. Proc.*, Atlanta, GA (1986).

[10] A. Luotonen and K. Altis, World-Wide Web proxies, Computer Networks and ISDN Systems 27 (September 1994).

[11] S. McCanne, V. Jacobson and M. Vetterli, Receiver-driven layered multicast, in: *Proc. of the ACM SIGCOMM '96 Conf.*, Stanford, CA (August 1996).

[12] T. McKinley, *From Paper to Web: How to Make Information Instantly Accessible* (Hayden, 1997).

[13] J. Mogul, F. Douglis, A. Feldmann and B. Krishnamurthy, Potential benefits of delta encoding and data compression for HTTP, in: *Proc. of the ACM SIGCOMM '97 Conf.*, Cannes, France (September 1997).

[14] L.B. Mummert, M.R. Ebling and M. Satyanarayanan, Exploiting weak connectivity for mobile file access, in: *Proc. of the 15th Symposium on Operating System Prinicples*, Copper Mountain, CO (December 1995).

[15] B.D. Noble, M. Price and M. Satyanarayanan, A programming interface for application-aware adaptation in mobile computing, in: *Proc. for the 2nd USENIX Symposium on Mobile and Location-Independent Computing*, Ann Arbor, Michigan (April 1995). Also as technical report No. CMU-CS-95-119, School of Computer Science, Carnegie Mellon University.

[16] B.D. Noble, M. Satyanarayanan, D. Narayanan, J.E. Tilton, J. Flinn and K.R. Walker, Agile application-aware adaptation for mobility, in: *Proc. of the 16th ACM Symposium on Operating Systems Principles*, St. Malo, France (October 1997).

[17] M. Podlipec, The XAnim home page. At the time of this writing, the document is available only through the World Wide Web at http://xanim.va.pubnix.com/home.html.

[18] L. Rabiner and B.-H. Juang, *Fundamentals of Speech Recognition*, Prentice Hall Signal Processing Series, A.V. Oppenheim, ed. (Prentice Hall, Englewood Cliffs, 1993).

[19] M. Satyanarayanan, *RPC2 User Guide and Reference Manual* (School of Computer Science, Carnegie Mellon University, October 1991).

[20] A. Smailagic and D.P. Siewiorek, Modalities of interaction with CMU wearable computers, IEEE Personal Communications 3(1) (February 1996).

[21] D.B. Terry, M.M. Theimer, K. Petersen and A.J. Demers, Managing update conflicts in Bayou, a weakly connected replicated storage system, in: *Proc. of the 15th ACM Symposium on Operating System Principles*, Copper Mountain, CO (December 1995).

[22] A. Waibel, Interactive translation of conversational speech, IEEE Computer 29(7) (July 1996).

[23] G.K. Wallace, The JPEG still picture compression standard, Communications of the ACM 34(4) (April 1991).

Brian Noble is an Assistant Professor in the Electrical Engineering and Computer Science Department at the University of Michigan. His research centers on the software supporting mobile computing systems, including networking, infrastructure, and end-system concerns. He received the Ph.D. in computer science from Carnegie Mellon University in 1998.
E-mail: bnoble@eecs.umich.edu

Mahadev Satyanarayanan is the Carnegie Group Professor of Computer Science at Carnegie Mellon University. The Coda and Odyssey systems for mobile information access have been developed over the last decade under his leadership. Earlier, he was a principal architect and implementor of the Andrew File System. He received the Ph.D. in computer science from Carnegie Mellon in 1983, after Bachelor's and Master's degrees from the Indian Institute of Technology, Madras.
E-mail: satya@cs.cmu.edu