3 Design Rationale

Before designing an interface that makes caching translucent, I must address the question of which aspects should be exposed to the user. Issues to consider include:

- What errors must be brought to the user's attention?
- What system behaviors cause confusion and why?
- What aspects of the system do users need and want control over?

I begin this chapter by discussing these questions and presenting aspects of caching that should be exposed to the user.

In sketching out a preliminary design of an interface to make caching translucent, I quickly realized that doing so could actually endanger the usability of the resulting system in significant ways. Although users may be less confused about the inner workings of the file system, they may spend so much time paying attention to those activities that they are unable to use the file system effectively. Thus, I continue this chapter with a discussion of the burdens imposed by translucent caching and then present five design principles intended to minimize those burdens. An overview of the activities and events that could be exposed to the user completes the chapter.

3.1 Windows of Translucence

In this section, I discuss those aspects of caching I considered exposing to users. I examine how to present errors to the user, how to reduce confusion caused by modal behaviors, and what aspects of the system users need to control.

3.1.1 Presenting Error Conditions

Contemporary computer systems, as a whole, fail to present errors to users in useful ways. What user cannot tell a story about a horrible error message they have encountered? In the process of writing this thesis, I have certainly run into more than my fair share. Although one might expect these sorts of errors from a Unix-based system, the error messages to which I am referring all came from supposedly user-friendly systems like Windows95 and Office97.

PowerPoint helpfully informed me that it could not print my document because either it had run out of memory or the printer was not on the network. Thankfully, I was disconnected from the network and had accidentally clicked the print button so it was an easy error for me to diagnose. However, in another situation, it could have been far more confusing. In all likelihood, the printing software received an error and handed that error off to the user rather than investigating the problem itself so that it could present the error with some potential solutions to the user.

Similarly, Microsoft Backup does not bother to determine whether the cause of its inability to write to the selected backup device was that (a) the device was write protected, (b) the network restricted access rights, (c) the device was full, or (d) the device name was not valid. Once again, the interface dropped the ball by not investigating the cause of the failure more thoroughly.

Experience has shown that, like these systems, neither AFS nor Coda present errors to users well. One user complained about AFS' habit of silently expiring tokens, making little effort to inform the user and allowing her to figure it out based upon seemingly unrelated symptoms of the problem. One very experienced Coda user reported a confusing behavior he had experienced recently. At the bottom of his bug report, he writes: "Eventually, by dumb luck, I realized that I had forgotten to clog [authenticate]. Boom, everything's fine, resolve works, I'm back in business. I think a better error message here would be nice" [post to cmu.cs.proj.coda.bugs on 17 July 97]. Clearly, the lack of tokens was not evident to either of these users. In this particular instance, an error reported to the requesting process may have been the right thing to do. In other situations, however, there may be no user process to which an error can be returned. Although both of these examples relate to token expiration, other less frequent errors could cause similar confusion. A successful interface for translucent caching will present error reports with appropriate amounts of detail to its users.

3.1.2 Rationalizing Behaviors

Coda users also experience confusion caused by fundamentally modal behaviors. Coda attempts to hide vagaries of network performance by quietly changing its behavior. When the system detects that network bandwidth is low, it automatically switches to weakly connected operation. When it detects that network connectivity has disappeared, it automatically switches to disconnected operation. By attempting to continue to operate even while network performance is substandard, Coda increases the availability of data and thereby increases usability. But because these transitions are hidden from the user, Coda decreases usability. Even Coda developers have expressed confusion over these silent transitions!

The zephyr logs[4] captured an excellent example of this confusion. A Coda developer and expert user wrote: "Help, why would venus [sic] claim a timeout on a directory when all

---

[4] Zephyr is an on-line chat system used by many members of the department to communicate. The Coda project uses a particular zephyr instance that all developers read at least occasionally.

of the servers are up? Further, you can ls -ld the directory, but not ls -l or ls…" [Coda zephyr instance, March 1996]. After chatting with him, it became clear that what had happened was that Venus had silently transitioned into weakly connected operation.[5] Because the directory in question could not be fetched within a certain time limit, Venus returned an ETIMEDOUT message. Because the user had not noticed the transition to weakly connected operation, he was baffled by the seemingly strange behavior he observed.

A second example, also captured in the zephyr logs, occurred in August 1997. Another Coda developer wrote to report strange behavior. Venus had inexplicably timed out during a compile into his Coda object area and had done so, albeit unpredictably, in the past. This time he had captured a Venus log file and wanted me to look at it. While we were trying to sort out this first strange behavior, we discovered that the log file, which he had saved into Coda, had zero length. These behaviors suggested that his Venus was operating weakly connected. Indeed, upon further examination, he determined that Venus had, unbeknownst to him, transitioned into weakly connected operation.

This incident, and similar ones not captured by zephyr, results directly from transparent transitions between different modes of operation. Modal behaviors were known to cause usability problems as early as 1973 [53] and the best solution, obviously, is to avoid them. However, because Coda provides these modal behaviors as a *feature* to support legacy programs that know nothing about adapting to changing network conditions, it cannot avoid them. Instead, Coda must take steps to reduce confusion caused by the behaviors. Research suggests that exposing the modes and their transitions to the user will help. Providing such feedback in other interfaces has reduced mode errors by as much as 70% [32]. Our interface for translucent caching should strive to make users aware of the current mode of operation and notify users when a transition between modes occurs.

3.1.3 Controlling Resources

The three most precious resources to the mobile client are cache space, network bandwidth, and battery power. Since issues of power resources are only indirectly related to file systems, I do not discuss them further. Instead, I focus my attention on cache and network resources.

3.1.3.1 Cache Space

During periods of disconnected operation, Venus exploits the cache to provide availability; during periods of weakly connected operation, Venus exploits it to avoid network-consuming demand fetches. Because the contents of the cache are so crucial to the success of these modes of operation, the current system exposes caching decisions to the

[5] The user was running a version of Venus that contained support for weakly connected operation. Because the network is overloaded, bandwidth frequently drops below the threshold that triggers weak connectivity.

user by allowing the user to prioritize file system objects. Venus then uses these priorities to make caching decisions (see Chapter 2 for details).

The current system offers the user 1000 different priority levels and requires her to prioritize individual files and directories. Users currently have difficulty determining all the files and directories they need to hoard. In fact, even expert users fail to hoard critical files occasionally. Users also have difficulty determining whether or not their hoard contents are available for use during an upcoming disconnected session. Thus, they tend to practice disconnecting one or more times prior to leaving their office.

Because preparing the cache for periods of disconnected and weakly connected operation continues to be an important duty, users continue to need a way to specify which file system objects are important. Further, they need feedback regarding which objects are currently available and which ones are not. An interface that makes caching translucent to the user needs to reduce the burden hoarding places on users and increase the feedback provided to users.

3.1.3.2 Network Bandwidth

Venus uses network resources to satisfy four important requirements, each vital to the integrity of the file system. Venus must:

1. Maintain cache consistency
2. Service fetch requests
3. Maintain cache equilibrium
4. Replay updates

Each of these activities is critical to the file system's integrity, but their relative importance to the user may vary over time. In this section, I explore each of these requirements.

3.1.3.2.1 Maintaining Cache Consistency

Venus' first responsibility is to maintain cache consistency. It does so by maintaining callbacks while strongly or weakly connected and by pure optimism while disconnected. While connected, the network costs associated with maintaining cache consistency are minimal. When a callback is lost, Venus simply invalidates the object or objects involved. While a callback is held, Venus must periodically verify that the server that granted the callback is up and able to maintain its promise. If Venus loses contact with the server, it must assume that the server is unable to maintain its callback promise; thus, the consistency of the object is suspect, but the object is not invalidated. Upon reconnection, Venus must validate its entire cache by reestablishing callbacks on all its objects. Because volume callbacks make the cost of validating the entire contents of even a large cache minimal (see Section 2.2.3), the expected *benefit* of exposing cache validation to the user is not high. For this reason, I chose not to explore this option.

3.1.3.2.2 Servicing Fetch Requests

One of Venus' most noticeable responsibilities is servicing fetch requests made on behalf of the user. The network costs associated with a request depend upon the size of the requested object and the speed of the network. Let us consider a simple example. Suppose the user has requested a 1 MB file. On a 10 Mb/s Ethernet, this file can be fetched in a second or two. Yet on a 14.4 KB/s modem, this file requires nearly 10 minutes to retrieve.

When considering this problem, I must remember three important points. First, a 1 MB file is not particularly large, especially with the increasing importance of audio and video. Second, because of the nature of a demand request, the user is most likely sitting around twiddling her thumbs until the fetch has completed. Finally, the user may have referenced the file indirectly or even accidentally. For example, the missing file could be a large postscript figure that is included in an area of a document the user is not currently working on. Rather than waiting for the figure to be fetched, the user may prefer to exclude the figure from the document temporarily. Because of the potential for excessively lengthy delays while Venus services demand requests, the user needs to have control over whether or not an object is fetched over a weak network connection.

3.1.3.2.3 Equilibrating the Cache

The third responsibility of Venus is to maintain cache equilibrium. Venus must balance the user's demand requests with the need to prepare the cache for an unexpected disconnection. The hoard daemon reequilibrates the cache periodically (during hoard walks) by recalculating the priority of each cached object and comparing those priorities to hoarded, but uncached, objects. This process ensures that the highest priority objects are cached.

The network costs associated with performing such a hoard walk can be quite substantial. Recall that a hoard walk occurs in two distinct phases. During the first phase (the status walk), Venus validates the status block of each cached object, refetching the status if necessary. Since each status block is only about 100 bytes and since most cached objects are valid, this phase is relatively inexpensive. The second phase, however, is a different matter. In this phase, Venus fetches data for each invalid or missing object identified in the first phase. For each of these objects, Venus could incur substantial network costs. These costs are directly proportional to the cumulative size of the file system objects that must be fetched. Because of the potential for extremely high costs associated with equilibrating the cache, the user needs to have control over the amount of data that will be fetched.

3.1.3.2.4 Reintegrating Updates

The final responsibility of Venus relevant to this discussion is the replaying of updates that have occurred during either disconnected or weakly connected operation. Objects may be reintegrated *in toto* or in chunks (as with trickle reintegration). Almost all types of updates can be replayed with only minimal network resources. The one exception is the `store` request, which requires that the file contents be sent to the servers. As with servicing fetch requests, the time required to transfer the contents of these files depends upon both the size of the file and the current network bandwidth. Again, these costs can be substantial for large files or slow links. Because weak connectivity only offers a limited amount of bandwidth, the user may prefer to delay propagation of updates to allow demand fetches or cache equilibration to have full use of those resources. Thus, users need to have control over whether reintegration occurs and, if so, how much.

3.2 Burdens of Translucence

The question I consider in this section is that of identifying threats to usability—in essence, the risks inherent in exposing aspects of caching to the user. The primary threat to usability for a mobile client of a distributed file system comes from critical cache misses—that is, objects that are critical to the user's work, but that are missing from the cache. Other threats result from the demands the system places on a user's valuable time and relatively limited cognitive capacity. In this section, I will discuss each of these potential threats.

3.2.1 Failure of Availability

A critical cache miss is the ultimate "sin" committed by the cache manager because it interrupts the user's train of thought and forces the user to change tasks or work around the missing object. A goal of this thesis is to reduce the likelihood of critical cache misses in ways that do not compromise the overall usability of the system.

3.2.2 Excessive Time Investment

The second threat to usability is the risk of requiring too much time to prepare for operating away from the office. The system requires time from the user in several distinct ways, including

- Learning to use the system and its interface
- Waiting for the system to service a variety of common requests
- Offering advice to the system to help it better provide high availability

User-assisted solutions to the high availability problem must remain cognizant of the time investment required of the user.

3.2.3 Overloading the User's Cognitive Capacity

The final threat to usability that I consider is the risk of overloading the user's cognitive capacity. This includes activities that require the user to remember more than short-term memory allows, that break concentration, or that demand immediate attention. Each of these "sins" centers the user's attention not on her own activities, but rather on the job of controlling the cache manager.

3.3 Principles of Translucence

A number of principles guided the design of the interface. In this section, I present each of these principles. At times, one principle contradicts another. The job of the interface designer then is to juggle these competing goals gracefully.

Principle #1: Above all, do no harm

The first principle summarizes the words of Hippocrates:

> The physician must … have two special objects in view with regard to disease, namely, to do good or to do no harm [12].

Rephrasing these apropos words, the designer must have two goals in developing interfaces, namely, to improve the lot of users, or at least not make it worse.

Principle #2: Work within the Unix model

Coda offers Unix transparency to applications accessing it. The system must remain consistent with this Unix model. As such, if the system can perform better with user assistance, it should be allowed to take advantage of that assistance. However, if no user is available to offer assistance, the system must continue on to the best of its ability. This requirement distinguishes my work from work in which designers freely assume the user is attending to the interaction.

Principle #3: Follow generally accepted principles of HCI designers

Interface designers need to follow generally accepted guidelines for building good human-computer interfaces. Although many guideline documents contain hundreds of pages, a few general heuristics provides more guidance in the "real world" simply because designers can remember them. These heuristics [31, 37] are:

1. *Simple and natural dialogue*: Minimize the information presented to that which is absolutely necessary and present that information logically.
2. *Speak the user's language*: Present information in words, phrases and concepts familiar to the user.
3. *Minimize the user's memory load*: Make information available when and where it will be used.
4. *Consistency*: Represent different words, situations, and actions differently.
5. *Feedback*: Keep users informed of the system's activities.
6. *Clearly marked exits*: Mark exits clearly so that users can easily back out of unwanted states.
7. *Shortcuts*: Offer accelerators to speed the interaction for the expert user.
8. *Good error messages*: Present error messages using plain language, indicate the problem precisely, and suggest possible solutions.
9. *Prevent errors*: Avoid error states whenever possible.
10. *Help and documentation*: Offer search capability, focus on the user's task, list necessary actions explicitly, and avoid verbosity.

Principle #4: Be unobtrusive

The third principle, a corollary of Principle #1, is simply to be as unobtrusive as possible. The user's goal is to write a paper or to implement an algorithm—not to babysit the cache manager. The system should alert the user to specific, important events and should allow the user to influence those events (when reasonable). The system must not demand immediate attention, must not be annoying in its interactions, and it must not "cry wolf". If the system fails in these goals, it risks not serving its users well.

Principle #5: Balance Costs with Benefits

By definition, a user-assisted approach to the problem of providing high availability requires that the user spend time on meta-level issues that would otherwise be spent on her primary goals. The user is willing to invest that time for the ability to access her data while mobile. Economics suggest that for this approach to be successful, the benefits must outweigh the costs. Otherwise, users will not waste their time. This principle serves to remind system designers of this fundamental law.

3.4 Overview of Translucent Caching

In this section, I provide an overview of which aspects of the system I expose to the user and how I reduce the costs of that translucence. I begin by discussing how to provide the user with feedback regarding both error conditions and modal behaviors. I then continue by discussing how to give users control over critical system resources—in particular, cache and network resources.

3.4.1 Feedback

The first design goal for the translucent caching interface is to provide the user with feedback about the system, in a way that does not interfere with the user's primary task. Users need two types of feedback. First, they need to be aware of error conditions (see Section 3.1.1). Second, they need to be aware of modal behaviors (see Section 3.1.2).

The distinction between error conditions and modal behaviors in Coda is blurred. Coda's goal is to hide error conditions, such as the lack of a network connection to the server, by transitioning to a new mode of behavior. These two aspects of Coda are very closely related. When an error condition causes a change in behavior, I consider that error a modal behavior rather than a true error.

One of the most common error conditions users experience is the expiration of their Coda tokens. When this happens, the system reduces the user's privileges to those of an unauthenticated user. In essence, the system has entered a new mode of behavior without reflecting that state in any obvious way to the user. In contrast, the translucent caching interface must provide the user with an indication of the current mode. This indication should distinguish between token expiration and activity pending tokens.

Other modes of operation that the interface must indicate to users relate to network connectivity. The interface must make users aware of the current mode of operation: strongly connected, weakly connected, or disconnected. If users are aware of the current mode of operation, their expectations for system behavior will change in subtle, but important, ways.

One class of true errors is related to storage allocation. If the system's local disk or Venus' RVM [47] space fill to capacity, Venus is likely to terminate ungracefully. When these space areas begin to get too full, the system needs to warn the user. When they become dangerously full, the system needs to warn the user more urgently. Similarly, if Venus' file cache becomes filled with hoarded objects, the user needs to be informed about the situation so that she can take measures to correct it.

This feedback, although important to usability, must not become intrusive or I risk making the system less usable than before. The system must make the user aware of system state without getting in the way.

3.4.2 Control over Critical Resources

The second design goal is to provide users with control over critical system resources. To reduce the burden this control places on users, the interface only exposes those aspects of resource management for which users need control. The two most critical resources users need control over are the cache and the network. In this section, I discuss the limited amount of control the interface offers to its users.

3.4.2.1 Cache Management Decisions

Coda users already have control over cache management decisions. The purpose of giving users this control is to allow them to ensure that their work is available in the case of a disconnection. Although the current system offers users this control, it does so in a way that compromises the usability of the system, particularly for novice Coda users. In essence, there is a mismatch between the information that the system needs and the information that the user can provide and vice versa.

The system compromises usability in two important ways. First, it does not speak the user's language. Users must specify their work at the level of files and directories, but often these files and directories are unfamiliar to the user because they are "owned" by a program that the user wants to use. For example, if the user wants access to the `latex` program, not only must they specify the `latex` binary but they must also specify the `virtex` binary as well as the necessary font and style files. Because users must supply hoard information at the level of individual files and directories, hoarding places an undue burden on them. Second, users are never quite sure whether or not their work is available. Because the cost of a missed file is substantial, this lack of confidence encourages users to perform multiple practice disconnections before actually leaving the office. To make caching translucent to the user while imposing only a minimal burden, I need to change both the granularity at which users provide hoard information and the way in which Venus communicates with users.

The interface addresses this issue by allowing users to define *tasks* and by providing feedback regarding the *availability* of those tasks for use during disconnected operation. To reduce the burden of hoarding, the system also automates a portion of the task definition procedure. A *task* contains user data and programs, as well as other tasks. User data is specified at the level of files and directories, because it is reasonable to assume that users are familiar with it. Programs, on the other hand, are specified at the level of executables. The system automates the process of defining programs by monitoring what file system objects a program accesses. Task-based hoarding is discussed in Sections 4.10 and 6.1.

Defining tasks is likely to be an error prone process. To help the user debug their definitions, the system must offer the user advice. This advice includes identifying objects the user should be hoarding but is not, as well as identifying objects the user is hoarding but should not be. By providing such advice, the interface further reduces the burden hoarding places on the user, thus reducing the costs and increasing the benefits.

Finally, the interface uses two techniques to increase user confidence regarding the ability to operate disconnected. The first, which has already been discussed, is to provide the user with feedback about the availability of tasks. The second, which eliminates the need to perform practice disconnections, is to provide a mode of operation that makes cache misses apparent to the user. While operating in this mode, the user can observe cache misses without having to physically disconnect from the network and without having to wait for the requisite timeout period. By eliminating the need for practice disconnections, I reduce the burden of preparing for disconnected operation.

3.4.2.2 Network Management Decisions

Weakly connected operation introduces a number of decisions that are very difficult to make automatically. In some situations, the user might want the system to do one thing; in other situations, the user might prefer the system to do the opposite. Because the user's preferences are not constant, heuristics will simply not work well in all situations.[6] To solve this problem, the interface makes certain network decisions translucent to the user. When the system needs to make one of these decisions, it issues an *advice request* to the user.

The first type of decision I expose to the user is whether or not to fetch a file over a slow network connection. Depending upon the size of the file and the speed of the network, a demand fetch could cause a substantial pause in processing. Depending upon the importance of the file to the user's work and the expected fetch time, the user may or may not want the system to initiate the fetch. My interface offers the user this control. To reduce the burden this control places on the user, I model the user's patience for fetching objects of various priorities and suppress the interaction when it appears to be of only marginal value. The *user patience model*, which appears in Section 6.3, helps make the interface less intrusive.

The second type of decision I expose to the user is the volume of data to fetch in the event of a hoard walk. During a hoard walk, the system may determine that it needs to fetch a substantial number of objects. If the client is currently operating weakly connected, the hoard walk could monopolize the network connection. Because the user may or may not actually need all these objects, my interface offers the user control over hoard walks. To reduce the burden, I employ the user patience model from above and also allow the user to request that the system stop asking about individual objects or tasks.

The final type of decision I consider exposing to the user is the volume of data to be reintegrated over a weak connection. Because weak connections limit the overall amount of network traffic, users need to have control over what portions of the CML are trickled back to the servers. Although not yet implemented, this aspect of the interface is described in Section 9.2.3.2.

---

[6] This is in direct contrast to a system that need only support programs running unattended. Programs have an infinite amount of patience so will not become frustrated if the system requires ten minutes to service a request. Further, people are far more flexible than programs. If one task is not available, a user is likely to choose a different one that is available instead.