# 9 Conclusion

Systems that exploit caching for availability will become more widespread in the years to come. Mobile computers, both laptops and personal digital assistants, require the ability to operate while disconnected or weakly connected. As users of these systems tire of managing their "cached" data by hand, more sophisticated support will be added. Our experience in Coda will help designers of these systems as they begin to fully support mobile users.

In this dissertation, I describe an interface that supports users of such a system. The interface presents the user with indicator lights, a metaphor borrowed from the dashboard of an automobile. These indicator lights keep users informed about the status of the system, but also allow them to request further information about a problem. The system allows users the option of controlling the use of network and cache resources. It also takes care to reduce the burden that these features impose on users.

In this final chapter, I first discuss the contributions of my thesis. I then outline areas for future exploration including ideas that need further validation, ideas for improving the interface, and ideas for extending the translucence. I conclude by completing the fictional story begun in the introductory chapter and with a few final remarks.

## 9.1 Contributions

The overall contribution of this work is the recognition and demonstration that translucent caching is a feasible and effective approach to making systems that cache for availability usable. I demonstrate this thesis with the design, implementation, and evaluation of a graphical interface that makes caching translucent to users of the Coda file system. At the next level of detail, my thesis makes the following contributions:

- *Introduction of the concept of translucent caching and recognition of its critical role in systems that exploit caching for availability.*

  Unlike caching for performance, caching for availability cannot remain transparent to users during periods of poor network connectivity. When a cache miss occurs while a client is operating disconnected, users suddenly notice that their files are not stored locally. Similarly, when a miss occurs while a client is connected via a modem, users notice substantial delays in accessing large files. This lack of transparency causes usability problems. The approach I take to address this problem is to systematically expose internal details of caching in a controlled way and to give the user the opportunity to influence how resources are used.

- *Design for a translucent caching interface*.

  This graphical interface makes caching translucent to users of the Coda file system. The system balances the tension caused by the need to preserve the Unix philosophy with the need to address the usability concerns discussed above. A key feature of Unix is the ability of programs to run unattended. Preserving this feature seems to preclude interacting with the user. My interface balances these conflicting goals: attentive users are given opportunities to control the system, yet programs running unattended (and inattentive users) are not significantly impeded.

- *Identification of techniques for reducing the burden of translucent caching*.

  Although translucent caching can improve usability, it also runs the risk of reducing usability. If a system using translucent caching is not careful, it could focus user attention on managing the cache and network, and detract from users' ability to get useful work done. This thesis examines the potential costs involved and introduces three techniques to reduce those costs. The first of these techniques, automatic program hoarding, reduces the cost of advising the system regarding what to cache. It automatically tracks references made by programs, differentiating between program data and user data to allow users to share program definitions between tasks. The second of these techniques reduces the cost of managing task definitions by identifying objects that the user may wish to add to or remove from a task definition. The third technique, modeling user patience, predicts whether users will be willing to wait for a file based upon that file's importance and the expected fetch time. The system then suppresses advice requests when the object can be fetched within the user's patience threshold. These techniques strive to reduce the burden that caching for availability places on users.

- *Implementation of a translucent caching interface and definition of a Translucent Caching API.*

  The interface, which was implemented within the context of the Coda file system, demonstrates the feasibility of these ideas. The system has recently been deployed to the Coda user community and we look forward to learning its benefits and limitations under real world conditions in the months ahead. The implementation has benefits beyond Coda, however. Because it was implemented external to Coda's cache manager, other highly available file systems could exploit the interface with a modest amount of effort.[37]

[37] The modifications necessary to implement this API within Coda's cache manager require approximately 2500 lines of code, and represent only a 3% increase to the overall size of Venus. Modifications to the graphical interface may also be required to customize the indicators to a different file system, but the basic events and ideas should be similar.

- *Development of an experiment to evaluate the effectiveness of the translucent caching interface in improving usability.*

  To evaluate the CodaConsole interface, I developed an experiment that examined questions of usability specific to translucent caching. The experiment was based upon a general evaluation technique common in HCI research, usability testing. As part of the experiment, I developed an on-line tutorial and exercises with which to teach participants about the interface and then observe them in action. Although the tutorial is obviously specific to the CodaConsole, the exercises could be employed, perhaps with minor modifications, to evaluate alternative approaches to the problem. Further, this work represents the first known application of HCI evaluation techniques to distributed systems research.

- *Results of a usability test performed on the CodaConsole interface.*

  The results of this test were striking: novice Coda users performed almost as well as expert Coda users! Users were presented with activities similar to those typically required of Coda users. They were asked to define and hoard tasks. They also experienced events that occur while using Coda, including disconnection, weak connection, cache misses, and authorization expiry. With the interface, users were able to understand what had happened and explain what actions they could take to resolve the various situations. This test validated my approach.

- *Feedback to the HCI community on my experience with usability testing.*

  My experience during the usability test lead to two observations of use to the HCI community. The first reveals the importance of verbal protocol to the process. During the analysis of the usability test, I realized that the transcripts of the verbal protocol contributed disproportionately to the usability findings. In fact, my analysis of the contribution of each piece of evidence to the usability findings shows that the verbal protocol was indeed the most important source of data. This result validates the use of verbal protocols in usability testing and explains the popularity of this technique with HCI evaluators. My second observation reveals the inadequacy of current techniques for estimating the number of users required during a test. The generally accepted rule for predicting how many users are required to uncover a certain percentage of findings did not work well for my usability test. In this test, the statistical prediction underestimated the number of users needed by 20-30% in precisely the range most critical to evaluators working in the field (e.g. six and fewer participants). This result contradicts previous experience with this prediction technique for which researchers found the model accurate to within a few percentage points. The thesis suggests an alternative predictor that worked well for my experimental data, but leaves open the question of its validity across other experiments.

## 9.2 Future Work

The work presented in this thesis represents an initial step toward translucent caching. The interface shows the feasibility of translucent caching and demonstrates improved usability. Additional areas of the system could, and should, be exposed to the user. As my work progressed, I also identified areas that deserve further attention, but that are beyond the scope of this thesis. In this section, I highlight these areas for future work.

### 9.2.1 Validating Aspects of the Interface

A number of important issues remain unvalidated. One such issue is our ability to predict the user's patience threshold. Another is the effectiveness of the heuristics used to offer users advice. A third is the effectiveness of my translation between user task priorities and Venus hoard priorities. Furthermore, although my thesis suggests that the interface will offer substantial benefits to users, the interface has not been validated under real-world conditions. In this section, I explain why each of these validations is necessary and what questions each validation should address.

#### 9.2.1.1 Heuristics for Offering Advice

The technique for offering users advice regarding the content of the task definitions, presented in Section 6.2, depends upon the success of the heuristics. These heuristics have not, as yet, been validated. Thus, an interesting experiment would be to measure the effectiveness of these heuristics. In particular, such an experiment should address the following two questions for each heuristic:

- How much information does the heuristic identify?
- How frequently does the user modify her task definitions to reflect the information provided?

The data necessary to answer the first question would be trivial to collect. The system need only be instrumented to record the information as well as the parameter values used by each heuristic, and send it to a data collection site. The data necessary to answer the second question, however, presents a small problem. The difficulty here is that currently it would be nontrivial to determine whether or not the user actually used the information (though see Section 9.2.2.3 below for one way to address this problem).

Although I have no doubt that these heuristics will identify files that users have forgotten to include in a task definition or that users forgot they included in a task definition, I am concerned about the burden this advice will place on users. If the heuristics identify too many false positives, users are likely to ignore the advice. By collecting usage statistics regarding the advice we offer users, we can answer questions about how much information these heuristics identify in practice and how frequently that information influences users' task definitions. Once we know how well these simple heuristics perform in practice, we may find that we need to explore more complex approaches to avoid false positives or to avoid presenting too much information to the user.

#### 9.2.1.2 User Patience Model

The model of user patience presented in Section 6.3 depends upon the hypothesis that a user's patience can be predicted with adequate accuracy. To date, this hypothesis has not been verified. An interesting experiment would be to collect the data necessary to support or disprove this hypothesis and, if appropriate, to then determine the shape of the curve.

Because individual differences could have an enormous effect on the resulting data, the data from individual users would need to be analyzed separately. Further, each user would need to experience a large number of weakly connected cache misses covering a wide range of priorities in order to provide a sufficiently clear picture of any potential model. Thus, this project requires a large number of users providing data over a lengthy period of time.

Assume for a moment that such a model has been verified and quantified. That model could offer benefits well beyond the realm of Coda. For example, suppose a user is searching for information on the Web, perhaps in Yahoo or in a database of newspaper articles. If the search engine can reliably predict the quality of a match, the client could use that information to predict whether or not the user will be willing to wait for a particular page. Although the use of such information would likely be different in the Web domain, that information has the potential to reduce the burdensome delays we have all experienced while using the Web.

#### 9.2.1.3 Task to Hoard Priority Translation

Alternative translations between task priorities meaningful to users and hoard priorities meaningful to Venus have not been compared experimentally. The current system uses the Fibonacci sequence to spread higher priority tasks further apart in the hoard priority space (see Section 6.3.2.1). The translation, although simple, has not been tested experimentally. An interesting experiment, then, would be to examine how different translations work in practice. Such an experiment should address the following questions:

- How many distinct tasks do users need? Similarly, how many well-separated tasks do users need?
- Should the translation be linear or nonlinear?
- How far apart in the hoard priority space must the highest priority tasks appear to allow Venus to make cache replacement decisions intuitive to users?

This experiment is intimately tied to the prioritized cache management policy used by Venus. Thus, a serious complication of this examination will be to differentiate between these two issues. Because these two issues are confounded, it may be appropriate to explore alternative cache management policies simultaneously.

#### 9.2.1.4 Real-World Conditions

The user test suggests that the interface will help users substantially, particularly new Coda users. However, these results have not been confirmed by actual usage experience. An interesting study would reexamine the questions presented in Section 7.2 under real-world conditions. Designing such an experiment requires careful thought to find ways to collect data that will actually answer these questions.

For example, one measure of a user's success in defining tasks is the frequency with which she experiences cache misses. Unfortunately, the system views one cache miss as being equivalent to the next, whereas users often do not. The severity of a cache miss is highly dependent upon the user's current goals. One method of addressing this complication is to request advice from the user. Figure 4.7 shows an example of how an experimenter might collect severity data. In this query, the system has requested information regarding the severity of a disconnected cache miss. With such data, the experimenter could estimate the effectiveness of a user's task definitions, looking at the frequency of critical cache misses as well as less severe ones.

### 9.2.2 Improving the Interface

The results of the user study identified many ways in which we can improve the usability of the interface. These were summarized in Section 7.6 and described in detail in Appendix F. In this section however, I present further suggestions for improving the interface. These suggestions are not a result of the usability test but simply of my reflections on the interface while writing this thesis.

#### 9.2.2.1 Parameters

One aspect of Venus, which the interface does not expose to the user, is the many parameters that Venus depends upon. Examples of these parameters include those controlling the user patience model, the prioritized cache management policy, the weak connectivity boundary, and the aging of records before reintegration. The parameters to Venus measure in the dozens. Some subset of these parameters could reasonably be exposed to the user with proper explanations of their purpose. We have never conducted experiments to determine the "best" parameter settings; instead, we have relied on reasonable performance in practice. Exposing these parameters to interested users will allow them to experiment, at least informally. They may discover settings that work better.

The proper place to expose these parameters is in a new tab of the *Control Panel*. On this tab, I would place another set of tabs that group related parameters. For instance, those related to the user patience model, to the prioritized cache management policy, to weak connectivity, and the like. Because Venus has reasonable defaults for each of these parameters, only interested users would need to explore this area of the interface.

### 9.2.2.2 Connectivity Information

The interface currently provides connectivity information (via the *Network* indicator light) at the wrong level of granularity (Comment 256). This indicator presents the information at the granularity of individual servers, but users want to know about files and directories. The interface would better serve users by tracking the connectivity of the individual volumes the user accesses and then presenting the connectivity of those volumes through an indicator light (perhaps replacing the *Network* indicator). The informational window associated with this indicator light would present this connectivity information in terms of the pathname at which each volume is mounted.

### 9.2.2.3 Incorporating Advice into Task Definitions

An obvious extension to the CodaConsole interface would be to connect the windows offering hoard advice to users with the actual program and data definitions used by the system. The basic idea would be to add a button to the Consider Adding and the Consider Removing windows (see Figure 4.11) that would either allow the user to add the identified object to a definition or to remove the object from all definitions. Such an extension would greatly simplify the process of incorporating advice. It also makes collecting the data necessary to validate these heuristics trivial (see Section 9.2.1.1).

### 9.2.2.4 Incorporating Ideas from Seer

Another interesting extension to the CodaConsole interface would incorporate the ideas used in Seer (see Section 8.1.5 and [26, 27]). The goal of such a system would be to reduce the burden that defining tasks places upon users. I see four approaches that could accomplish this goal.

The first method defines each Seer cluster as a task. This method completely removes the burden of defining tasks from the user, a major benefit. However, this benefit comes with a high cost. Because clusters are not always meaningful to users [27], the *Task* indicator also becomes meaningless. Users can tell that a cluster is not available, but they cannot determine what that unavailability implies. Users would also be unable to prioritize clusters for the same reason. The fact that clusters are not meaningful to users represents a fundamental problem with this approach. For this reason, in the following paragraphs I explore other ways to incorporate Seer with the CodaConsole. These other methods provide a lesser benefit in terms of the requirements for either the cache space or the user's attention, but do not sacrifice the meaningful feedback offered by the *Task* indicator. Of course, should extensions to Seer eventually produce clusters that are meaningful to users, those clusters could easily be substituted for the CodaConsole's user-defined tasks.

The second method exploits Seer clusters to offer advice to the user regarding their definitions. By examining clusters and comparing them with definitions created by users, the system could suggest forgotten files. Again, because the cluster may include files unrelated to this task (at least in the user's mind), the suggestions may be less than helpful to users. The success of this approach depends heavily upon what percentage of files within a cluster are meaningful and intuitive to users. The current version of Seer is unlikely to achieve an accuracy high enough to support this method [24].

A third alternative would be to use the user's definitions to give feedback to the clustering algorithm. This feedback could easily [24] be incorporated as hints to the Seer clustering algorithm, with what is called an external investigator [27]. At the very least, such feedback could prevent Seer from dividing a single task into multiple clusters. It also has the potential to produce clusters of files that are meaningful, though perhaps noisy, to users. This approach represents a compromise between the automatic nature of Seer and the feedback offered by the CodaConsole. It also allows the user to choose the level of interaction. With no help, she would get Seer's current level of performance, with some feedback from the CodaConsole. After defining tasks, the CodaConsole would provide feedback completely meaningful to the user and Seer would presumably perform even better.

The fourth approach simply uses Seer's clusters to automatically extend the task definitions created by the user. The system could then forgo offering advice to the user about their task definitions. The lack of meaningful clusters would not be problematic because the user would continue to communicate with the system in terms of her defined tasks. The burden that task definitions place on users would be minimal because the definitions would no longer need to be exact.

These four approaches represent a range of possibilities. At one extreme, Seer is silently incorporated into the CodaConsole as tasks. This approach is the ideal because it completely removes the burden of defining tasks from the user, but it is currently impractical because the clusters that Seer produces are not meaningful to users. The other approaches reduce the burden that defining tasks places on users, but do not remove that burden entirely.

### 9.2.3 Extending the Translucence

The interface presented in Chapter 1 focuses on making caching translucent to the user. Other aspects of Coda must also be made translucent to the user. One such aspect is repairing object replicas that have become inconsistent because of optimistic replication. Another is allowing the user control over the amount of data that trickles back to the servers during reintegration. Although these aspects of the interface have not been implemented, a preliminary design[38] has been developed consistent with the interface presented earlier.

---

[38] Because the designs presented here represent "paper designs", the screens shown in Figure 9.1 and Figure 9.2 appear with a white background, making them visually distinct from the screens that have already been implemented.

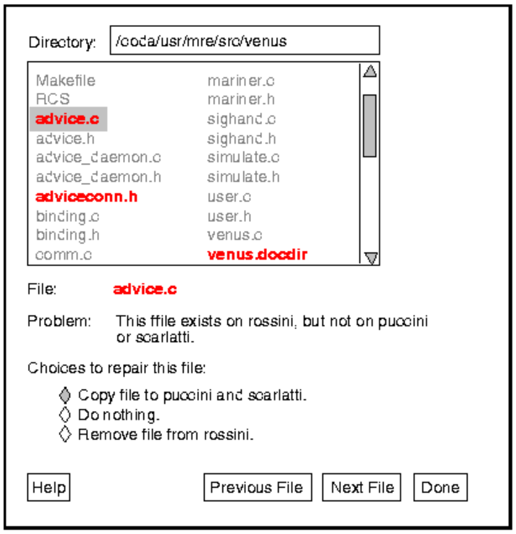
### 9.2.3.1 Repairing Objects in Conflict

Ideally, users could invoke the tools for manually repairing conflicting replicas (see Section 2.2.1 and [28]) of an object via the CodaConsole interface. The preliminary design for incorporating repair into the CodaConsole interface hooks this function into the *Repair* indicator light. The difference is that the user would now be able to double-click on the name of an object in conflict (in Figure 4.14, the user would be able to double-click on /coda/usr/mre/thesis/dissertation). After doing so, the user would see a screen similar to those shown in Figure 9.1. It would explain the cause of the conflict and give the user some choices as to how to repair the conflict. In the case of true conflict (Figure 9.1b), the user would also be able to look at the different versions of the file.

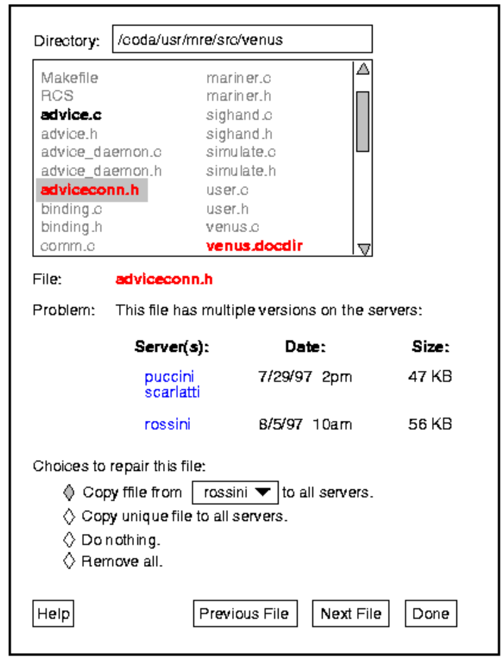
### 9.2.3.2 Controlling Trickle Reintegration

Trickle reintegration (see Section 2.2.3 and [34]) uses the network wisely without human intervention. Attentive users, however, should have the ability to control what data gets reintegrated. The preliminary design for giving users this control within the CodaConsole interface hooks into the *Reintegration* indicator light. Once implemented, the *Reintegration Information* window shown in Figure 4.13 would be replaced by that sketched in Figure 9.2(a). From this screen, the user would be able to select a set of subtrees to be reintegrated. In addition, the user would be able to double-click on the name of a subtree needing to be reintegrated to view a display of the CML, as sketched in Figure 9.2(b). From this screen, the user would be able to indicate how many of the updates should be reintegrated. Once reintegration has begun, a window showing the name of the current subtree and a progress meter would appear.

A further extension, one that would make this a task-based interaction, would allow the user to select which task's updates should be reintegrated. Such a version would be far more complicated to implement because of the need to ensure a proper ordering on the updates and is not discussed further here (for more information on the complexities involved, see [33]).



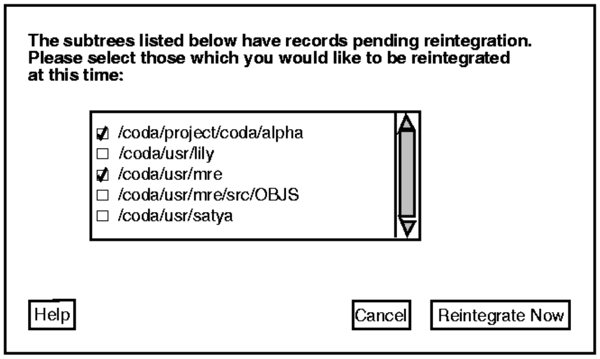

(a) remove/update conflict



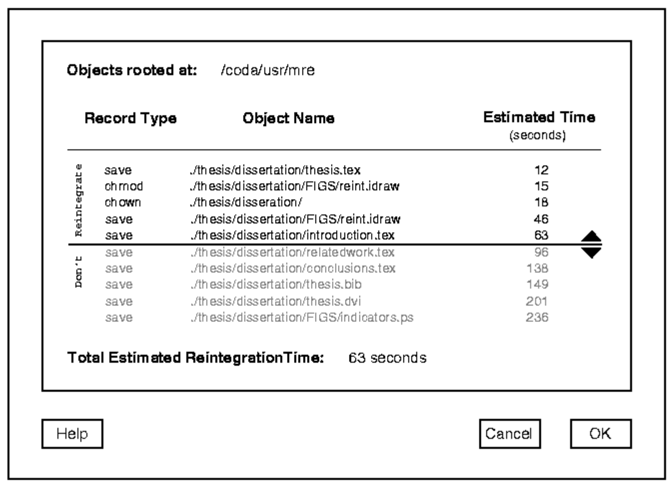

(b) update/update conflict

**Figure 9.1: Repairing Objects in Conflict**

This figure shows the preliminary design of the repair module for the CodaConsole interface. Windows similar to these would appear when the user double-clicks on the name of an object in conflict in the *Repair Information* window of Figure 4.14. View (a) shows the screen that would allow users to manually repair a directory with a remove/update conflict. In this case, the file exists on `rossini`, but not on `puccini` or `scarlatti`—perhaps one user deleted it while communicating with `puccini` and `scarlatti` but another user updated it while communicating only with `rossini`. View (b) shows the screen that would allow users to repair an update/update conflict manually. In this case, users who were operating partitioned from one or more servers both updated the file. Thus, the version of it stored on `puccini` and `scarlatti` differs from the one on `rossini`. In these drawings, objects whose names appear in red are in conflict, those whose names appear in black have fixes pending, and those whose names appear in gray are not in conflict. Servers whose names appear in blue represent hyperlinks that allow the user to examine the content of individual replicas. The next and previous object buttons take the user to the next or previous object in conflict. The done button begins the actual repair.




(a) Proposed Reintegration Information




(b) Controlling the Amount of Data with a Subtree

**Figure 9.2: Reintegration Interface Paper Design**

This figure shows the preliminary design of the reintegration module for the CodaConsole interface. View (a), the proposed replacement for the window shown in Figure 4.13, allows the user to select which subtrees (volumes) will be reintegrated. View (b), which would appear after the user double-clicked on a subtree shown in view (a), allows the user to control how many updates within a subtree (volume) get reintegrated. The line in the middle of the listing represents the reintegration barrier [34]. Those updates appearing above the barrier will be reintegrated; those below will be postponed. The user can move the barrier up or down using the arrow widgets. The total estimated reintegration time for those objects selected to be reintegrated is shown near the bottom of the window.

## 9.3 Final Remarks

Let me now return to our story of Peer, Yoni, Alex, and Leah. When last we heard of our intrepid users, they were having difficulty understanding what the system was doing and why. Alex had asked Yoni to look into a newly released interface add-on that he thought addressed the problems they were experiencing.

> Yoni found the documentation about the new interface and discovered that it did address this problem and more. He downloaded a demo version of the interface and found that it used a small number of indicator lights that kept him informed about the system's activities and about error conditions. Now, as he used his system remotely, he could tell when his data was available to his coworkers at the home office. He also discovered numerous unexpected benefits. With the new system, it became obvious when his tokens expired, when servers crashed, and when a task became unavailable. He quickly and easily discovered objects that became in conflict. He found the advice offered by the system about his task definitions to be useful in discovering files that he had inadvertently forgotten to hoard, and also in cleaning out task definitions that he no longer needed. After a few weeks of running the demo version of the software, he recommended to Alex that he purchase copies for their entire group. [39]

As distributed file systems and mobile computing continue to increase in popularity, users are going to begin experiencing situations similar to those described in this thesis. Users will observe failures. They will experience vastly different network latencies, bandwidths, and costs. The systems will have to adapt, and sometimes that adaptation will require changing the standard behaviors.

Systems will need to expose failures and behavior changes to users or else greatly reduce the usability of the resulting system. These details will need to be carefully presented to users in a meaningful way. This thesis shows one path these systems can take.

---

[39] And, yes, they all lived happily ever after.