# 8 Related Work

This chapter presents work related to this thesis. It is divided into three sections. The first section presents systems that also address what has become known as the hoarding problem. The second section discusses computer interfaces that use a dashboard metaphor. The third section presents work on open implementation because this interface can be considered an example of such a system.

## 8.1 Hoarding

A number of other systems address the hoarding problem. In addition to Coda, these systems include Little Work, FACE, Briefcase, Transparent Analytical Spying, Seer, and Laputa. The last of these systems, Laputa, describes an application-specific solution to the hoarding problem. The remaining systems each describe their approach for addressing the general hoarding problem.

To aid in understanding these different systems, Figure 8.1 presents a taxonomy for these different approaches. The taxonomy includes a number of different categories.

*Granularity of objects*: Describes the granularity with which each system specifies the files and directories that need to be hoarded in preparation for disconnected operation

*Prioritization*: Specifies whether users can prioritize hoarded objects and, if so, how many distinct priorities they can use

*Identification of objects*: Describes how each system determines which objects need to be hoarded in preparation for disconnected operation

*Distinguish application data from user data*: Specifies whether the system makes a distinction between application and user data

*User Interface*: Describes the interface to the system that is presented to the user

| Characteristic | Little Work | FACE | Briefcase | Coda (original) | TAS | Seer | Coda (revised) |
|---|---|---|---|---|---|---|---|
| | **Mechanisms** | | | | | | |
| Granularity of Objects | | | | | | | |
| File-based | ✓ | ✓ | ✓ | | (✓) | | ✓ |
| File-based + Meta | | | | ✓ | | | ✓ |
| Command-based | | | | (✓) | ✓ | | ✓ |
| Task-based | | | | | (✓) | ✓ | ✓ |
| Prioritization of Objects | None | None | None | 1000 | None | None | 88 |
| Identification of Objects | | | | | | | |
| User Responsibility | | ✓ | ✓ | ✓ | ✓ | | ✓ |
| History Horizon | LRU | | | LRU | | | Extended |
| Spying | | | | Manual | Automatic | | Automatic |
| Semantic Distance | | | | | | ✓ | |
| Distinguish (application vs. user) | | | | | | | |
| Doesn't | ✓ | ✓ | ✓ | ✓ | | ✓ | |
| User Responsibility | | | | (✓) | | | |
| Heuristics | | | | | ✓ | | ✓ |
| User Interface | None | Command | Graphical | Command | Graphical | Command | Graphical |
| Extent | | Basic | Basic | Basic | Basic | Minimal | Extensive |
| Feedback | | None | Minimal | None | None | None | Extensive |

**Figure 8.1: Taxonomy of Caching for Availability Mechanisms**

This table summarizes the characteristics of different approaches to the hoarding problem. The original Coda system refers to that described by Kistler [23]; the revised Coda system refers to that described in this thesis. The taxonomy begins by examining the granularity with which objects must be specified in each system. Some systems require the user to specify individual files and directories, sometimes augmented with meta-information (e.g. similar to that described in Section 2.3.2). Other systems allow the user to specify programs while still others offer support for tasks. It continues by specifying whether the system allows objects to be prioritized, and if so, how many distinct priorities are supported. It then examines how objects are identified for hoarding: by the user, by a history-based mechanism (perhaps LRU or an extended history horizon), by spying on file references, or by semantic distance. It then indicates whether the system distinguishes between application and user data, either internally using heuristics or externally with the help of the user. Finally, it describes the user interface provided by the system, including the type of interface (command-line or graphical), the extent of the interface (*minimal* allows the user to warn the system of an impending disconnection; *basic* supports user-assisted hoarding; *extensive* supports further interaction), and the amount of feedback provided. Checkmarks shown in parentheses indicate that users can do this if they choose, but that the system does not explicitly support or require this activity.

Let me make this taxonomy concrete by reviewing where the original Coda system, as described in Chapter 2, fits into this scheme. The granularity with which Coda specifies objects to be hoarded is at the level of file system objects, but it allows users to provide meta-information for directories. The original Coda system supported up to 1000 distinct priorities for hoarded objects. The user was responsible for hoarding objects, although, because the hoarded data was stored directly in the client's cache, the system also cached objects that had been accessed recently. The system did not distinguish between user and application data, except insofar as such a distinction was made by users within their hoard profiles. It presented a basic command-line interface to the user with no feedback about the state of the system.

## 8.1.1 Little Work

*Little Work* [14, 15] represents the baseline solution to the hoarding problem. This system offers absolutely no explicit support for users of disconnected operation. Because the system uses an LRU caching algorithm, users simply prime the cache before disconnecting by running the programs they expect to use during the disconnected session. Users evidently do this using their shell as the system provides no user interface. The authors have also added a *fetch-only* mode of operation that allows them to fetch any missing objects while connected over a weak link.

## 8.1.2 FACE

FACE [3] is the first system that considered exploiting user assistance to increase data availability. They implement a secondary cache, which they call a *stash*. While using a disconnected client, users access data out of this stash. In contrast, Coda uses a unified cache for both performance and availability.

Alonso and his colleagues suggest a number of techniques to identify files that need to be stashed. The first of these is a set of permanently stashed files, a list of which is kept in a well-known location such as `~/.stashrc`. Their second suggestion requires writing a tool that parses Makefiles to identify files that a given Makefile requires to run. Their third suggestion is approximately equivalent to Coda's spy tool. Their fourth suggestion requires instrumenting applications.

Their prototype appears to have implemented just two commands supporting user-assisted hoarding, neither of which was listed in their original suggestions. The first of these adds a file to the stash; the second removes a file from the stash. The authors offer no indications as to which of their suggested interfaces were implemented, nor do they offer any indication as to the success of user assistance in improving availability.

### 8.1.3 Briefcase

*Briefcase* [41, 50], a feature of Microsoft's Windows95, allows users to put copies of their documents into an electronic briefcase. In essence, Briefcase is simply a user-friendly term for what is, effectively, a cache. Users explicitly place documents in this cache before disconnecting. During disconnected operation, users modify cached documents. Upon reconnection, the user can request that the cache be synchronized with the local copy of the data. During synchronization, the system uses the timestamp associated with both the primary copy of the file and the cached copy of the file to determine which one to keep (the most recently updated file wins). Users can override this choice manually.

Briefcase assumes users are synchronizing a laptop with a desktop computer. It assumes that all necessary applications are resident on both machines. Thus, the problem is one of synchronizing user data only. It offers no support for indirect references to files. For instance, suppose a Word document includes a link to an Excel spreadsheet. If the user places that Word document into her Briefcase, the system is not smart enough to also include the Excel spreadsheet. To make matters worse, updates to the Excel spreadsheet are not reflected as updates to the Word document that includes a link to them. Thus, Briefcase is not smart enough to realize the primary copy of the Word document changed. On a simple test, I managed to get my Word document to become inconsistent with my Excel spreadsheet even after repeated attempts to force Word to update the imbedded object. My use of Briefcase produced not a single error or warning message about the actions I had taken. Briefcase is a simple-minded solution that addresses only a limited form of the hoarding problem, and is not yet ready for naïve users.

### 8.1.4 Transparent Analytical Spying

*Transparent Analytical Spying*[36] [52] applies prefetching research [51] to the problem of hoarding. This work effectively extends Coda's spy utility by providing a simple user interface to control spying, as well as a hoard selection interface to control what gets cached prior to disconnecting.

This system works by observing file accesses made by executing programs. For each program, it builds a set of access trees. Each access tree corresponds to a single execution of a given program (and its descendant processes). Because these access trees are maintained over the long term, they must be coalesced to save space.

Automatic program hoarding, described in Section 6.1 is also motivated by Tait and Duchamp's prefetching research. The primary difference between my subsystem and their Transparent Analytical Spying work is in their use of access trees. The access trees, which were critical to the success of the original prefetching application, offer no advantage to their system or my subsystem presently. Such trees could, in theory, help to prune the set

[36] I call this system *Transparent Analytical Spying* because that is the closest the authors come to naming this system. Kuenning [26] calls this system the *Spy Utility*. I chose not to use this term because of the conflict with Kistler's *spy* utility [22].

of file objects hoarded, but would require further information to do so without significant user input. For this reason, I chose to simplify the design by maintaining simple access lists.

This system also introduces the notion of *generalized bookends*. The user specifies when the system should start and stop spying. In essence, this system offers a graphical interface to a more sophisticated version of the spy tool (see Section 2.3.3) available in the original Coda system. At that time, she can specify a name, called a bookend, for the set of references made during the spy session. These bookends allow the system to group top-level activities. Bookends serve a purpose similar to that of tasks in my system.

Before disconnecting, the user requests that the system hoard her data. The system then displays a hoard selection window that allows the user to select the bookends that should be hoarded. In the same window, the user can select individual programs and specify additional files and directories.

Transparent Analytical Spying is the first system to present a graphical interface to the user of disconnected operation, though the operations are very basic. The interface provides no feedback to the user regarding the availability of the selected data or anything else.

The level of user assistance varies depending upon the interest of the user. The authors' goal as far as user assistance is concerned seems to be that of allowing optional assistance. Uninterested users would be able to accept the defaults and be reasonably happy. Interested users would be able to customize the hoarding selections and improve their disconnected availability. The authors have published no studies showing whether they met this goal, or examining the usability of their graphical interface.

## 8.1.5 Seer

*Seer* [25, 26, 27] applies multidimensional clustering to the problem of hoarding. This is the first system that requires almost no intervention by the user. The results are quite impressive.

This system identifies related files and directories by exploiting temporal locality of reference. The authors observe that related files and directories are generally accessed close together in time. They define the *semantic distance* between two file system objects, A and B, based on the number of opens that occur between the open of A and that of B. Using the semantic distance between files, Seer clusters related files into groups.

Before disconnecting from the network, the user must warn the system—this warning is the only user interaction required by Seer. Upon receiving this warning, the system assigns priorities to objects according to the clusters in which they appear. All elements of a cluster are assigned the same hoard priority, using an algorithm described in [26].

Seer offers two unique results. The first is its ability to identify the user's working set with almost no assistance from the user. The second is its ability to minimize the amount of cached data associated with each cluster. Together, these results offer significant utility to users of disconnected operation.

Seer's weakness is its inability to build clusters that are meaningful to its users. In fact, the authors explicitly list this characteristic as a disappointment:

> "The clusters produced by Seer often have contents that are surprising to us, either by including apparently unrelated files or by separating a single project into a few clusters rather than the single grouping that would correctly represent it." [27]

This weakness is unfortunate because it makes incorporating these results into the CodaConsole interface difficult. Ideas for doing so, however, will be presented in Section 9.2.2.4.

### 8.1.6 Laputa

The approach taken by Laputa (described in [49]) can only be described as application-specific hoarding. The goal of this system is to provide disconnected operation to software developers. The authors exploit process information readily available in their software development environment, Marvel [18], to determine what files should be hoarded to support a given activity specified by the user. The status of their implementation is unclear and their experience with the system has not been published to date. Because this system is specific to a particular application, I do not include it in my general taxonomy.

## 8.2 Dashboard Metaphor

One would expect the dashboard metaphor to be a common metaphor for exposing failures to users. Because of the popularity of automobiles, it has been widely deployed and accepted by consumers, at least in the United States. To date, I have found surprisingly few systems that exploit this metaphor for use with untrained computer users.

One particularly good example however, called *Net.Medic* [2], monitors Internet performance for naïve Internet users. The system claims to monitor network performance, isolate the bottleneck, diagnose the cause, and even correct certain problems. It presents information to naïve users in the form of a dashboard, using the familiar red, yellow, green color scheme.

The Net.Medic dashboard follows the dashboard metaphor more closely than does the CodaConsole. The gauges presented by Net.Medic closely resemble gauges of an automobile. The interface presents both analog gauges for network speed and throughput and digital gauges for retrieval time and connection time. Net.Medic does not concern itself with continuous feedback to users regarding network performance. Users must explicitly examine the monitor. Because of this design assumption, their system does not worry about minimizing screen real estate during normal operation.

Two other examples, Dashboard95 (a product of Starfish Software) and Front Panel [30], apparently provide interfaces that help users maneuver through a crowded desktop. They give quick access to common tasks, such as electronic mail and calendars. They keep important information at hand, such as the status of inboxes and the current date and time. Front Panel, at least, allows users to customize a set of *bins*. These bins contain the user's current work tasks. The main difference between the dashboard metaphors used by Dashboard95 and Front Panel, and the metaphor used by Net.Medic and CodaConsole is the focus. The goal of the former systems is to present quick, easy shortcuts to common tasks (the function of the lighting and environmental controls on and to the right of the steering wheel). The goal of the latter systems is to present error conditions to users (the function of the display panel behind the steering wheel).

A Web search uncovered one additional use of the dashboard metaphor, a dashboard interface builder [1]. This system allows developers to quickly build dashboard interfaces to specialized astronomy equipment. The system has apparently been used at both the Keck Observatory and UCO/Lick Observatory.

The dashboard metaphor will increase in popularity as people recognize its benefits. The metaphor has been introduced in a number of specialized interfaces. As people take notice of these specialized interfaces, they will begin incorporating the metaphor into more general computer systems. Office97 already includes extensive control options similar to those of Dashboard95 and Front Panel. It also uses a small number of indicator lights to provide users with feedback about the status of print jobs and of battery power.

# 8.3 Open Implementation

Computer science undergraduates are taught in Software Engineering 101 (or perhaps even earlier) to use a "black box abstraction". They learn to design an interface to their libraries and hide the implementation behind that interface. This technique offers substantial benefits in reducing the complexity of the resulting system. Unfortunately, this reduction comes at a high cost.

When the designer begins implementing the "black box", certain design choices must be made. Making these decisions requires the implementer to make assumptions about the client of the interface. Kiczales provides a nice example[20]. He considers a windowing system and the impact of certain implementation decisions on one particular application, the display portion of a spreadsheet. The basic problem is that the windowing system's implementation assumed only 25-50 heavy-weight windows (e.g. an xterm) would be used—not a thousand or more light-weight windows (e.g. a cell of a spreadsheet). Once that assumption had been hard-coded into the "black box", the spreadsheet implementer could no longer use this windowing system.

Kiczales argues:

> It is impossible to hide all implementation issues behind a module interface. Some of these issues are crucial implementation-strategy decisions that will inevitably bias the

> performance of the resulting implementation. Module implementations must somehow be opened up to allow clients control over these issues as well [20].

He continues by advocating an approach that separates the primary interface to the functionality of the module from a secondary interface that allows the client programmer to tune the underlying implementation of that module for a particular application if necessary. He states three goals for an open implementation. The client programmer should be able to:

- Use the module's primary functionality alone when the default implementation is adequate;
- Control the module's implementation strategy when necessary; and
- Deal separately with functionality and implementation-strategy decisions.

A good example of an open implementation is Kaashoek's work on the Exokernel [17]. In this system, the designers point out that operating systems define the interface between applications and hardware. They also point out the unfortunate consequences of hard-coding the implementation of operating system abstractions, including the fact that applications cannot take advantage of domain-specific optimizations. The goal of their kernel was to give applications control over how machine resources are used while protecting applications from one another. They provide a low-level interface to the hardware, and build operating system abstractions on top of it in libraries that are linked into the application. Client programmers can pick those abstractions most appropriate to their domain or they can start from scratch if no off-the-shelf libraries will work.

My work falls into the category of an open implementation of the cache for a distributed, highly available file system. My work differs from these and similar approaches in that the client for whom I am opening the implementation is actually the end user. Although this makes certain issues critically important (specifically, the user interface), the problems and goals of my system are closely related to those described above. In fact, I can restate Kiczales's argument as it applies to a distributed file system for which network resources are scarce or expensive as follows:

> It is impossible to hide all resource usage decisions behind a "black box" file system. Some of these crucial decisions will inevitably bias the performance of the resulting system. Mobile file sytems must somehow be opened up to allow users control over these decisions.

This thesis describes a way to give users control over these decisions, without sacrificing usability. The goals that Kiczales set out also apply to my system. Specifically, the user should be able to:

- Use the system "as is" when the default policies are adequate;
- Control the policy decisions when necessary; and
- Deal with the file system and the policy decisions in largely separate ways.

One additional point that he does not make explicitly, but that he implies by the secondary (and optional) interface is that the system's policy mechanism should be as unobtrusive as possible, remaining in the background when not in active use.

## 8.4 Summary

This chapter has described work related to that described in this thesis. Many people have looked at the hoarding problem, though no one has looked at ways to make caching translucent to users. Surprisingly few systems exploit the dashboard metaphor to expose failures to users, though this metaphor is becoming more common. Finally, this work represents an open implementation at a level higher than that traditionally considered.