4 Interacting with the User

The unifying theme of the interface, called the *CodaConsole*, was borrowed from another complex system, the automobile. Cars communicate with their drivers through a dashboard interface. Typically, the dashboard contains a number of indicator lights as well as a small number of gauges. The purpose of the lights is to grab the driver's attention and, thus, alert them to the current state. The purpose of a gauge is to offer more detailed information. For example, when the engine temperature gets dangerously high, a light might illuminate to alert the driver to the problem. The driver could then consult the engine temperature gauge to determine the severity of the problem.

Dashboard indicator lights have been reasonably successful and widely adopted. The strength of this interface lies in its ability to keep the user informed while remaining unobtrusive and allowing the driver to concentrate on the primary task of getting from here to there. But, even with its overwhelming acceptance, it is far from perfect and has at least two serious deficiencies.

The first deficiency results from not providing sufficient warning to fix certain problems before they can cause serious damage. In particular, when the oil light illuminates, the driver must stop immediately or risk serious engine damage. Although this problem may have developed suddenly, it is more likely to have been caused by a slow oil leak that went undetected for a long period of time. This lack of sufficient warning has given the dashboard interface the nickname *idiot lights*, supposedly because you are an idiot if you wait until the indicator lights before noticing a problem. This is the unfortunate consequence of a poorly designed implementation of what is an otherwise well-conceived idea.

The second of these deficiencies is its extremely limited vocabulary. A car is a complex piece of machinery. The dashboard is simply not large enough to offer an indicator and gauge for every conceivable ill. Because of this constraint, automobile designers have had to balance two competing goals. Their first goal is to alert the driver to problems. Their second is to give the driver more detailed information. They have wisely made the first goal their top priority. The number of indicator lights on the dashboard of a typical car is larger than the number of gauges. Thus, when a subsystem that only has an indicator light (and not a gauge) experiences a failure the interface offers an extremely limited vocabulary for expressing that problem. An indicator light generally expresses just three pieces of information. These three bits of information, regarding the subsystem that it monitors, are:

operating normally: indicator remains off in its steady state

needs attention: indicator remains on in its steady state *indicator operational*: indicator turns on briefly upon ignition

Because of this limited vocabulary, the user may be completely unable to determine the severity of a problem without either taking the car to a service center or getting under the hood.

Dashboard indicator lights make a complex mechanical device translucent to its user. They allow naïve drivers to recognize problems easily. I borrowed the idea for the CodaConsole interface because my goal was synonymous. The fact that so many users (particularly in the United States) are already familiar with indicator lights only makes the idea more appealing. After borrowing the basic idea however, I then refined it to address the deficiencies described above.

The CodaConsole interface strives to warn the user of impending problems so that she can take a proactive role in their solution or can actively prepare for the inevitable. Obviously, some problems occur instantaneously and the system cannot always forewarn the user. Still, the goal is to offer warnings whenever possible. Second, I have increased the size of the interface's vocabulary. Rather than expressing just one of two stable states, each indicator expresses four stable states—not operational, fully operational, warning, and critical. In addition, the indicators are dynamic. If the user wants more information about a problem, she can double-click on the indicator light and a "gauge" will appear. The overriding goal of the interface was to make caching translucent without imposing too great a burden on the user. In this chapter, I present the design and implementation of the interface.

4.1 The Indicator Lights Interface

The indicator window itself consists of nine indicator lights[7] as shown in Figure 4.1. The nine indicators and their functions are

Control Panel: allow the user to customize the behavior of the interface

Tokens: alert the user to problems related to authentication

Space: alert the user to problems related to storage space

Network: alert the user to problems related to network connectivity

Advice: alert the user to opportunities to give or receive advice

Hoard Walk: alert the user to the state of the hoard daemon

Reintegration: alert the user to the state of the reintegration daemon

Repair: alert the user to file system objects that are currently in conflict

Task: alert the user to changes in the availability of hoarded tasks; allow the user to define and hoard tasks

[7] Technically, the first of these, the *Control Panel*, is not an indicator light because it does not actually indicate events to the user. For simplicity, however, I will refer to it as an indicator light.

These nine indicators appear in a small window that measures about 1” x 2” on the user’s display. These indicators require a modest amount of screen real estate, a resource that is precious to users, during normal operation. Because the expected benefit of the indicators outweighs its cost in screen real estate, I expect the user will keep this indicator window visible at all times. Note, however, that while the user investigates a problem indicated by the interface, the screen real estate requirements increase. This change in resource consumption is appropriate during periods when the user is actively engaged with the interface [8].

Each indicator light can signal one of four states about the subsystem it monitors:

- Operating normally
- Developing a problem or experiencing a noncritical problem
- Experiencing a critical problem
- Indicator not operational or status unknown

The first three of these states are color coded to one of three urgency levels: normal, warning, and critical. By default, the colors associated with these urgency levels are green, yellow, and red. I chose this color scheme to make it easy for users to remember since green, yellow, and red correspond to the colors used to signify similar meanings in many situations in the United States—most notably traffic signs and signals. This color scheme is not without its own difficulties. For instance, some users may be using a monochrome monitor, others may be color blind, and not all cultures use green, yellow, and red for these meanings. For these reasons, the user can easily customize the color scheme used to represent these three urgency levels. In fact, the user can forego the use of color and instead use different grayscale levels to signify the different urgency levels. Figure 4.1 shows two views of the indicator window: one in the default green-yellow-red color scheme and one in a monochrome scheme.[8]

(a) Color Scheme (b) Monochrome Scheme

Figure 4.1: Indicator Lights

This figure shows two views of the indicator lights. The indicator lights give users a peripheral awareness of system state. View (a) shows the default green-yellow-red color scheme. View (b) shows a monochrome scheme. In both views, the *Tokens* and *Network* indicators are shown with a warning urgency level and the *Space* and *Task* indicators are shown with a critical urgency level. All other indicators are normal.

---

[8] A color supplement to this thesis is available at http://www.cs.cmu.edu/Reports

In this section, I present each of the nine indicator lights and their associated informational windows. For each indicator, I begin by describing its purpose and the meaning of its color changes. I then describe the layout and operation of each window associated with that indicator. Some of these indicators (*Control Panel*, *Tokens*, *Advice*, *Hoard Walk*, and *Task*) have multiple supporting windows. Others (*Space*, *Network*, *Reintegration*, and *Repair*) have just a single one. I begin the presentation with the *Control Panel* indicator and all of its supporting windows, and continue through the other indicators in the order shown in Figure 4.1.

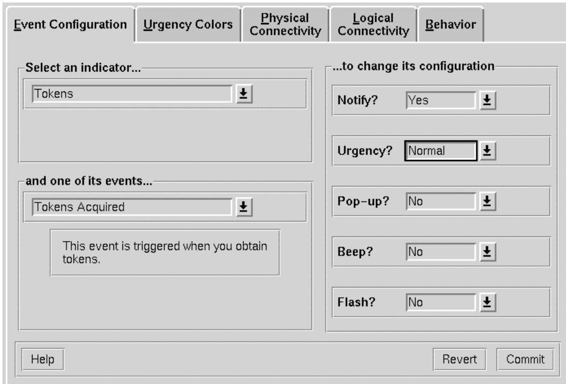
4.2 Control Panel

The first indicator light, a special case, does not actually indicate problems to the user (it is always green). Instead, it gives the user a way to control the handling of events, the colors of the urgency levels, the physical and logical connectivity to servers, and other behaviors of the interface. In the sections that follow, I will explore each aspect of the control panel.

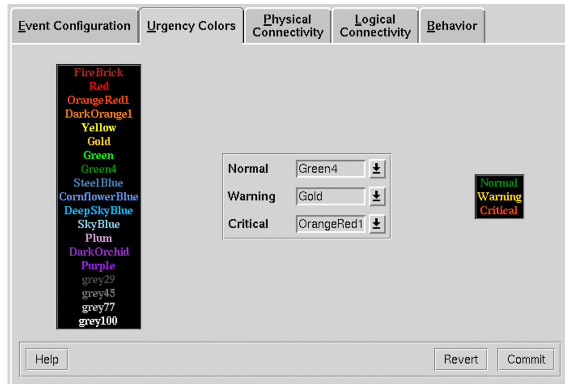
4.2.1 Event Configuration

When the user double-clicks on the *Control Panel* indicator, the *Event Configuration* window (one tab of the control panel window) appears. From this window (shown in Figure 4.2a), the user can customize event notification. In particular, the user can specify whether or not she wants to be notified of a particular event. If so, she can specify how the system will indicate its occurrence to her. Her notification options range from automatically popping up the appropriate window to quietly changing the color of the given indicator light.

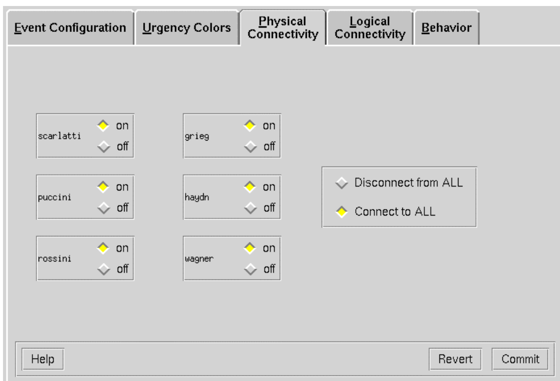
The layout of this window organizes the events by the indicator that signals them. Thus, only those events associated with the selected indicator can be chosen, and selecting a different indicator changes the list of events that can be chosen. Once an event is selected, the user can change its configuration using the right-hand side of the window. The user must first decide whether or not she wishes to be notified of this event. If she chooses not to be notified, all of the other selections are grayed out. If she chooses to be notified, she must then define the urgency of the event and how she wishes to be alerted to this event. The minimum (and default) action the system takes to alert the user of an event is to change the color of the event's indicator to reflect the urgency of the event (e.g., red, yellow, or green). In addition, the user can customize the alert to pop up the appropriate window automatically, to beep, and/or to flash the indicator light on and off a few times.



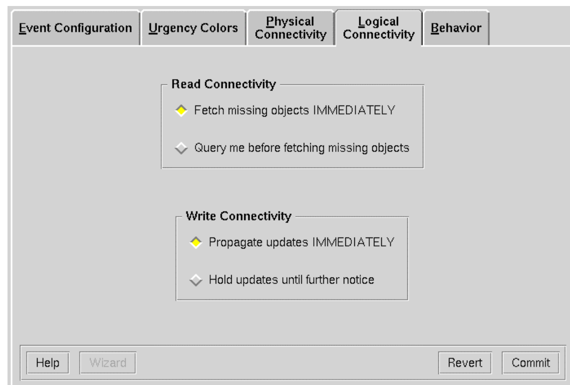
(a) Event Configuration Tab



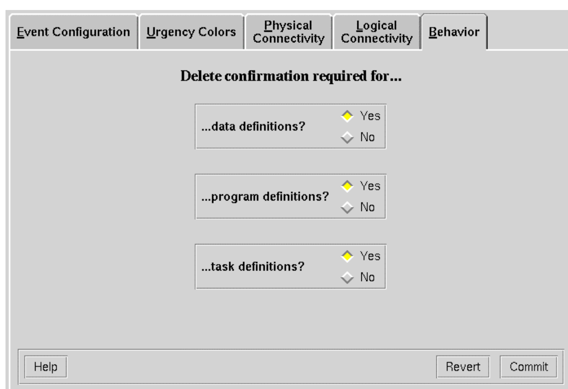
(b) Urgency Colors Tab



(c) Physical Connectivity Tab



(d) Logical Connectivity Tab



(e) Behavior Tab

Figure 4.2: The Tabs of the Control Panel

This figure shows the supporting windows that appear when the *Control Panel* indicator is double-clicked. From these windows, the user can control various aspects of the interface. The *Event Configuration* tab, view (a), allows the user to control the way events are notified. The *Urgency Colors* tab, view (b), allows the user to customize the color scheme used to indicate the three levels of urgency. The *Physical Connectivity* tab, view (c), allows the user to control connectivity to each Coda server. The *Logical Connectivity* tab, view (d), allows the user to control the servicing of cache misses and the propagation of updates while remaining fully connected to the network. The *Behavior* tab, view (e), allows the user to disable confirmation dialogue boxes.

4.2.2 Urgency Colors

If the user clicks on the *Urgency Colors* tab of the *Control Panel*, she will see the window that appears in Figure 4.2(b). From this window, the user can choose what colors the interface will use to indicate each of the different urgency levels. By default, the normal urgency level is indicated by the color green, the warning level is indicated by yellow, and the critical level is indicated by red. The user has a choice of about a dozen colors and four different levels of grayscale.

The layout of this window shows three vertical sections. The leftmost section shows a list of color names.[9] Each name is shown in the named color on a black background. This section gives the user an idea of how the color will look in the indicator light window. The rightmost section shows the three colors selected to represent the three different urgency values. This section allows the user to see how easily the three chosen colors can be distinguished from one another. The middle section provides widgets that allow the user to change the color representing each urgency level. As the user plays with her color choices, they are reflected in the sample indicator area on the right. The actual indicator lights are not updated until the user commits her choices by selecting the Commit button.

4.2.3 Physical Connectivity

If the user clicks on the *Physical Connectivity* tab of the *Control Panel*, she will see the window that appears in Figure 4.2(c). This window allows the user to toggle the network connection to individual servers, as if the interface could simply unplug a wire running between this client and each individual server. Obviously, this window does not control the physical network cables. Instead, it instructs the communication package that is linked into the Venus binary to drop packets to and from the indicated server. Controlling the connectivity in this way allows us to stop communication without physically unplugging the network connection; furthermore, it gives a finer level of control than physically unplugging the network connector. By default, the connection to each server is turned on.

Controlling file system activities at this level is an advanced feature, not one I would ideally expect novice users to master. It is, however, useful for performance reasons, when operating weakly connected, to disconnect from all but one server in each replication group. This tab could also be extended to allow the user to control the available bandwidth to or from a server by issuing the appropriate instructions to the communication package, though this has not been implemented.

---

[9] Although it would be simpler to make the drop-down list boxes contain the colored listing shown on the left, this is not currently possible in the `tix` widget library.

4.2.4 Logical Connectivity

If the user clicks on the *Logical Connectivity* tab of the *Control Panel*, she will see the window that appears in Figure 4.2(d). This window allows the user to control read and write connectivity to the Coda servers. *Read connectivity* is the ability to service cache misses—in essence, the ability to read data from the servers. *Write connectivity* is the ability to propagate updates—in essence, the ability to write data to the servers. Logically, the system can operate connected or disconnected with respect to each of these forms of connectivity. When a client is operating disconnected, it is both read and write disconnected. When a client is operating connected, it is both read and write connected. When a client is operating weakly connected, it could be operating read and write connected (the default) or it could only be operating connected with respect to one of these activities. This window gives the user control over which types of file system requests the server handles and which the client handles.

The ability to control read connectivity to the servers allows users to test their preparation for an upcoming disconnected session without the inconvenience of waiting for the connections to servers to timeout. The ability to control write connectivity to the servers allows users to improve performance during periods of poor network connectivity (or poor server performance), as well as when their activities involve numerous updates that are temporary in nature (e.g., `make venus` to build the Venus binary, followed immediately by `make clean` to remove object files). Controlling file system activities at this level is an advanced feature, not one I would initially expect novice users to master.

4.2.5 Behavior

If the user clicks on the *Behavior* tab of the *Control Panel*, she will see the window that appears in Figure 4.2(e). This window allows the user to turn off the confirmation dialog boxes that appear when she deletes a program, data, or task definition. By default, such deletes require confirmation. This tab could also be extended to give users control over other behaviors, though this has not been done.

4.3 Tokens

The second indicator light, labeled *Tokens*, alerts the user to problems related to authentication. When the indicator is shown in green[10], the user knows that she has valid authentication tokens. If the user double-clicks on the *Tokens* indicator, the window shown in Figure 4.3(a) appears. This window informs the user as to when her tokens expire and allows her to reauthenticate or destroy her tokens. When the indicator light

[10] Throughout this thesis, I assume the user is operating with the green-yellow-red color scheme.

(a) Tokens Valid (b) Tokens Expired

(c) Tokens Expired and Reintegration Pending

Figure 4.3: The Windows of the Tokens Indicator

This figure shows three windows associated with the *Tokens* indicator light. If the indicator light is shown with a normal urgency level and the user double-clicks on the indicator, then the window shown in view (a) will be displayed. View (b) will be shown when the indicator is at a warning urgency level and view (c) will be shown at a critical urgency level.

changes to yellow, the user knows that her tokens have expired. If she double-clicks on the *Tokens* indicator at this point, the window shown in Figure 4.3(b) will appear allowing her to reauthenticate. If the user does not have valid tokens and either a file system request or a reintegration is pending, the indicator light turns red. Double-clicking on it will cause Figure 4.3(c) to be displayed. This window describes any pending activities and allows the user to authenticate.

An alternative design would use yellow to indicate that a user's tokens were *about* to expire and red to indicate that they *had* expired. This design, however, loses the indication that an activity is actually waiting for the user to obtain tokens. I chose not to use this alternative design because of a problem that I had observed among Coda users. Occasionally, a user would reconnect to the system, but fail to authenticate. Without tokens, the client could not reintegrate their updates with the servers. Users would then become confused because their updates had not yet appeared on the servers. In addition, though, I felt that there was nothing inherently bad in not having tokens and so changing the indicator to red just because the user's tokens had expired seemed too alarmist. It seemed more appropriate to warn them and then turn the indicator red when the user's activity was stalled because of a lack of tokens. Upon further reflection, I would probably chose the alternative design if I were to redesign the system today. People seem to find these meanings more intuitive than those used in the current design.

4.4 Space

The third indicator light, labeled *Space*, alerts the user to problems related to a lack of space. The space indicator light monitors three distinct areas. The first area is the file cache. The user is alerted when the percentage of cache space devoted to hoarding file system objects exceeds either the warning or critical thresholds. The second area is the local disk. The user is alerted when the available space remaining on the local disk is less than that required for the remaining Venus cache.[11] The third area is RVM, an internal area required for proper Venus operation.

The window displayed when the user double-clicks on the *Space* indicator contains three gauges as shown in Figure 4.4. The length of the gauge is proportional to the size of the cache, disk, or RVM, respectively. The length of the colored bar represents the amount of space currently in use; the color of the bar represents the urgency of the problem. Green means there are sufficient resources to continue normal operations. Yellow alerts the user to a potential problem. Red alerts the user to a critical problem. The help window explains what corrective action, if any, the user can take to resolve the problem. The contents of this help window changes as the *Space* indicator state changes.

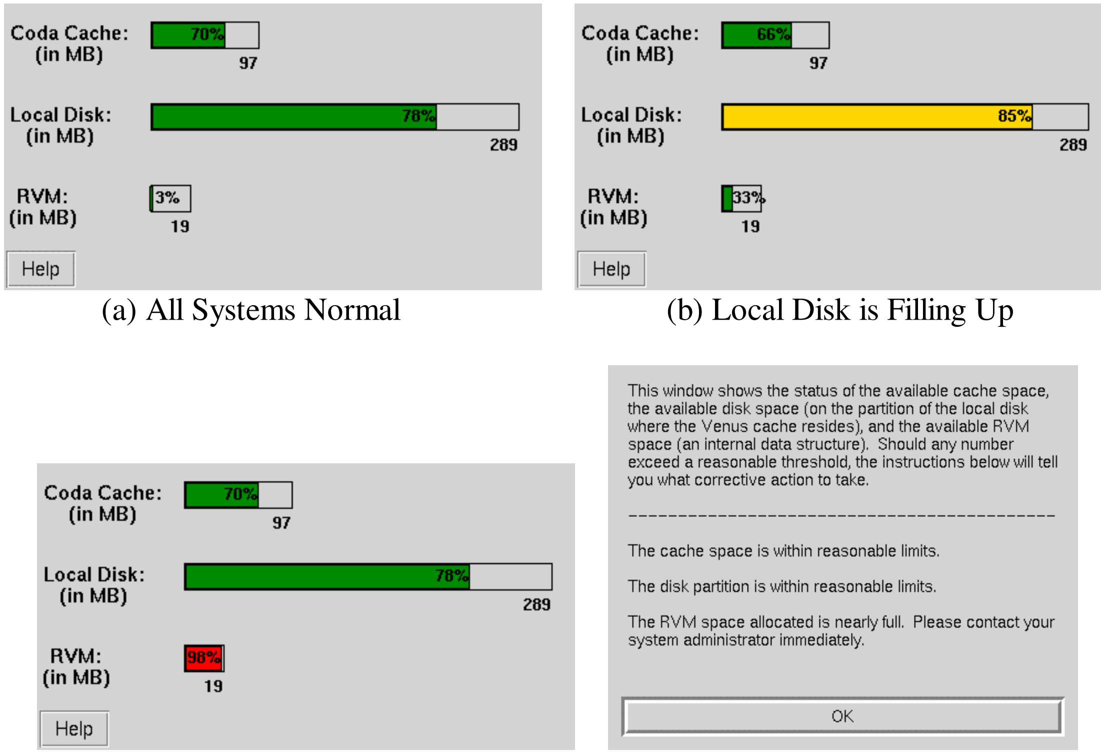
4.5 Network

The fourth indicator light, labeled *Network*, alerts the user to problems related to network connectivity. When this indicator is shown in green, the user knows that Venus is operating strongly connected to all Coda servers from which it has cached data. When this indicator is shown in yellow, the user knows that Venus is operating weakly connected to at least one Coda server, and that it is *not* disconnected from any servers. When this indicator is shown in red, the user knows that Venus is operating disconnected to at least one Coda server.

Double-clicking on the *Network* indicator light will cause a window similar to those in Figure 4.5 to appear. This window displays a gauge representing the current bandwidth available to each Coda server. The label to the left of the gauge is the name of the server. The length of the bar represents the maximum Ethernet bandwidth. The length of the colored bar represents the current bandwidth to this server as a percentage of that maximum. The color of the bar represents the state of connectivity to this server: strongly connected (green), weakly connected (yellow), or disconnected (red). If the name of the server is shown in gray, then the user has artificially manipulated the network connection to this server using the *Physical Connectivity* tab of the *Control Panel*. The user may click on the Control Panel button to bring up this tab to resolve the situation.

---

[11] For example, Venus is typically "allocated" space for its cache on a local Unix partition shared with other programs and users. If a misbehaving program or a malicious (or naïve) user fills that partition and leaves Venus inadequate space for this cache, Venus behavior is undefined. At worst, Venus will fail. At best, its cache performance will suffer. Alerting the user to this problem allows the user the opportunity to resolve the problem and either prevent Venus from failing or improve its cache performance.



(a) All Systems Normal

(b) Local Disk is Filling Up

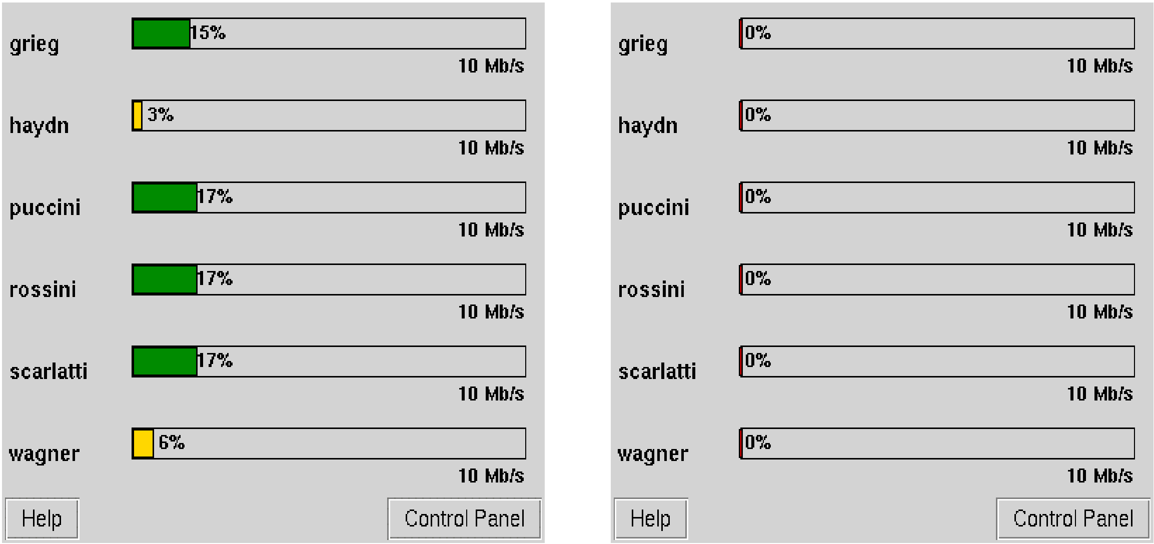
(c) RVM Space Almost Full

(d) Help Window for (c)

Figure 4.4: Space Information Window

This figure shows three views of the *Space Information* window as well as its help window. View (a) shows the *Space Information* window when all three areas are within normal limits. The three gauges in this view are all green. View (b) shows the window when the local disk is beginning to fill up. Because there is still space available, the local disk gauge is yellow (as would be the *Space* indicator light). View (c) shows the window when the RVM space is almost full. Because there is almost no RVM space available, its gauge (and the indicator light) would be red. View (d) shows the help window that would appear if the user were to click on the *Help* button of view (c). This window explains that the user should contact their system administrator to resolve this problem.

Currently, this window shows one gauge for each server of which the client is aware. As the number of Coda servers increases, the content of this window becomes lost to the user. This window should be modified to list only those servers the user is actively accessing. If the user is accessing more servers than can fit on the window, the interface should allow them to be scrolled. Neither of these extensions would be difficult. Further, the gauge shows the network bandwidth as a percentage of the maximum Ethernet bandwidth, a metric that is not meaningful to the user. Rather than showing the relative percentage of Ethernet bandwidth (e.g., 3%), the interface could show the absolute bandwidth estimate (350 Kb/s). Neither of these modifications to the interface would be difficult.



(a) Client operating weakly connected (b) Client operating disconnected

Figure 4.5: Network Information Window

This figure shows two views of the *Network Information* window. View (a) shows that this client is operating weakly connected to `haydn` and `wagner`, but strongly connected to all the other Coda servers. The gauges for `haydn` and `wagner` are yellow; those for the other servers are green. The *Network* indicator light would be yellow in this situation. View (b) shows that this client is operating disconnected to all servers. The gauges show a thin sliver of red. The *Network* indicator would also be red.

The distinction between strong and weak connectivity is made based upon comparing the current network bandwidth estimate to a threshold. If the current bandwidth is above the threshold, the system considers itself to be strongly connected; if it is below, the system considers itself weakly connected. Ideally, users should have the ability to control the value of this threshold, though novice users should not be required to manipulate it. Such an extension is discussed in Section 9.2.2.1.

The behavior of this indicator, as described here, is probably not what I would design today. The problem is that the indicator changes state before it is really appropriate to do so. For example, as soon as the system realizes that a server has crashed, the indicator turns red. However, because most Coda files are replicated, the fact that a single server has crashed is not terribly important to the user. It would be more appropriate to change the state of the indicator when a volume has transitioned into the disconnected state (cf Figure 2.1). The informational window would then need to expose the pathnames of volumes operating disconnected (and weakly-connected) as well.

4.6 Advice

The fourth indicator light, labeled *Advice*, alerts the user to pending requests for advice from Venus and of hoard hints. At critical decision points during operation, Venus may request advice from the user. These requests frequently relate to network usage. At other times, Venus may notice anomalies in the user's hoard database and may alert the user to them by way of a hint.

When the user double-clicks on this indicator light, the window shown in Figure 4.6 appears. It has two sections. The top section, labeled Advice Needed, contains a list of advice requests. The bottom section, labeled Advice Offered, contains a list of hoard hints. Each type of request and each type of hint are marked with either a warning or critical urgency level. By default, requests are marked as critical only when a thread is blocked waiting for a response. All other requests and all hints are alerted at the warning level. The color of the indicator light is red if any critical requests are pending user attention, and yellow if any noncritical requests or hints are pending attention. It is green if there are no requests or hints outstanding.

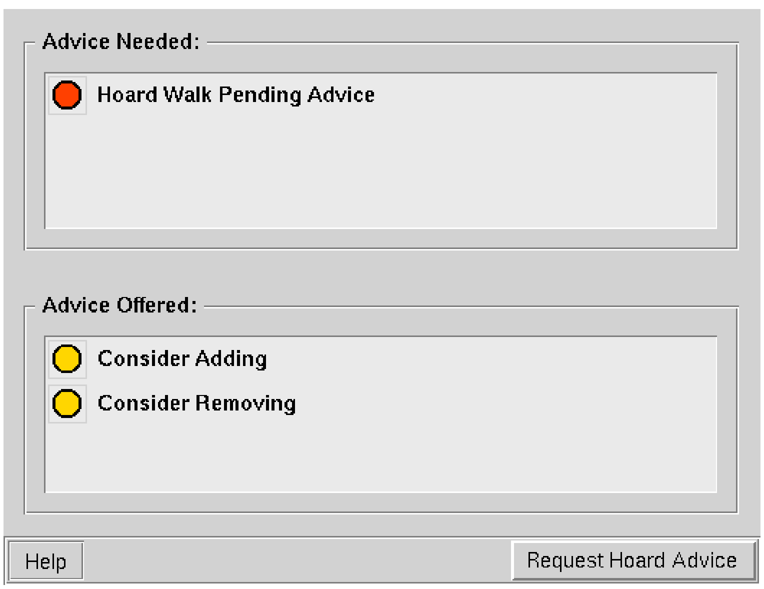


Figure 4.6: Advice Information Window

This figure shows the *Advice Information* window. It is displayed whenever the user double-clicks on the *Advice* indicator light. The top portion of the window, labeled Advice Needed, contains queries posed by Venus. By answering these questions, the user may influence the behavior of Venus. The bottom portion of the window, labeled Advice Offered, contains hints offered to the user regarding the content of the cache and of the hoard database. The circles to the left of each entry indicate the urgency of the request.

4.6.1 Requesting Advice from the User

Venus requests advice from the user in two situations. The first occurs when a cache miss occurs. The necessary advice comes in three flavors depending upon the current network connectivity. If Venus is disconnected from the servers of this object, then the request shown in Figure 4.7 arrives. The purpose of this advice request is to obtain evaluation data from the user. If, instead, Venus is operating weakly connected to all of the servers of this object, then the request shown in Figure 4.8 arrives. The weak-miss query allows the user control over demand requests. Finally, if Venus is strongly connected to the servers of the missing object and the user has put the system in *read disconnected* mode, then the request shown in Figure 4.9 arrives. The read disconnected miss query makes caching translucent to the user, making cache misses observable events. The second type of advice request occurs during hoard walks when Venus is operating weakly connected. During the course of a hoard walk, Venus may need to fetch substantial amounts of data—not all of which is strictly necessary from the user's point of view. The request shown in Figure 4.10 gives the user control over the quantity of data that is fetched during the data walk phase of a hoard walk. Each of these advice requests is discussed in the following sections.

4.6.1.1 Disconnected Cache Miss Query

The disconnected cache miss query arises from a cache miss that occurs while Venus is operating disconnected. The purpose of this query is to determine the severity of a given cache miss. Because the severity of a miss is highly dependent upon the user's current activities and goals, the system cannot determine this information automatically. Instead, the system requests the user's assistance. An important use of such data would be to evaluate the CodaConsole interface under conditions of actual use, as will be discussed in Section 9.2.1.4.

As shown in Figure 4.7, the query contains the name of the missing object as well as the name of the program requesting that object. The user is asked to estimate the severity of the miss using a scale. The user may indicate uncertainty if she cannot determine the severity of the miss. The user can also indicate that she is testing a hoard database for a future disconnected session. Finally, the user is given an opportunity to provide additional information, though this is optional.

4.6.1.2 Weakly Connected Cache Miss Query

The weakly connected cache miss query (see Figure 4.8a) results from a cache miss that occurs while Venus is operating weakly connected from all of the servers of the missing object. The purpose of this advice request is to allow the user control over network resources when those resources are scarce. Once again the system tells the user the name of the missing object and of the program requesting that object. In addition to this information, the system also tells the user the estimated fetch time of the object. The system queries the user to determine whether or not to fetch the object.

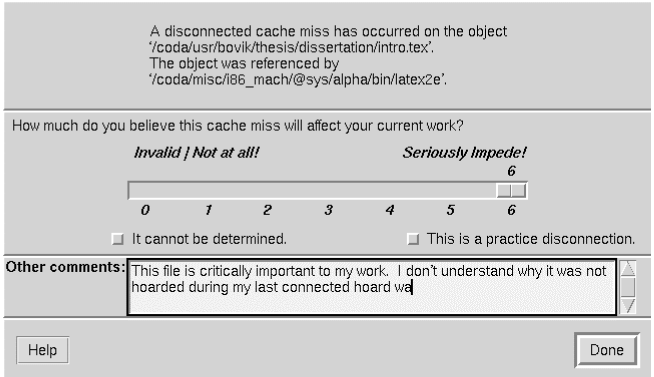


Figure 4.7: Disconnected Cache Miss Questionnaire

This figure shows a query that results from a disconnected cache miss. The top portion of the window contains the name of the missing object and of the program requesting it. The middle portion allows the user to indicate how much the cache miss will affect the user's work. The bottom portion allows the user to provide any additional comments she might have regarding this miss.

A complication of this design arises because the user may be unavailable to answer the request. Perhaps the user has stepped away from her laptop to get dinner. This is where the Unix paradigm is important. If the user is unavailable to answer a request, she will want the system to have made progress during her absence. Imagine her frustration if she returns from dinner only to find a dialog box blocked waiting for an answer! To alleviate this problem, the system times out after a reasonable period (currently, about 15 seconds). Because the system cannot determine whether the user is considering her options or is unavailable, it presents the window shown in Figure 4.8(b). If the user was simply slow in making a decision, she will notice the prod and can request that the system continue to wait (at which point the window shown in Figure 4.8c is displayed). If the user was, indeed, unavailable, this prod will also time out and the system can take the default action (currently, to fetch).

An obvious design alternative is to initiate the fetch and provide the user with the name of the object, the name of the requesting program, a progress meter, and an abort button. Such interfaces are extremely common, and given current network access characteristics, more appropriate. In the current implementation of Venus however, aborting a fetch (once it has been started) is not possible without tearing down the entire RPC connection. The ability to abort a fetch in progress would certainly be useful, but it would require a fair amount of programming for which the research contributions are minimal. Further, in the future, I expect networks to charge per packet (or per byte). Once money is involved, users will demand not only the ability to abort a request but also the ability to approve charges before they are incurred. For these reasons, I choose to explore the "ask permission" design alternative and leave the "beg forgiveness" one for future extension.

An interesting extension would be to recognize when the system is running unattended. For instance, after the third time a request for advice times out, the system might decide that the user must not be available. When the user is unavailable, the system could stop asking for advice and act on the best available information. The difficulty here is

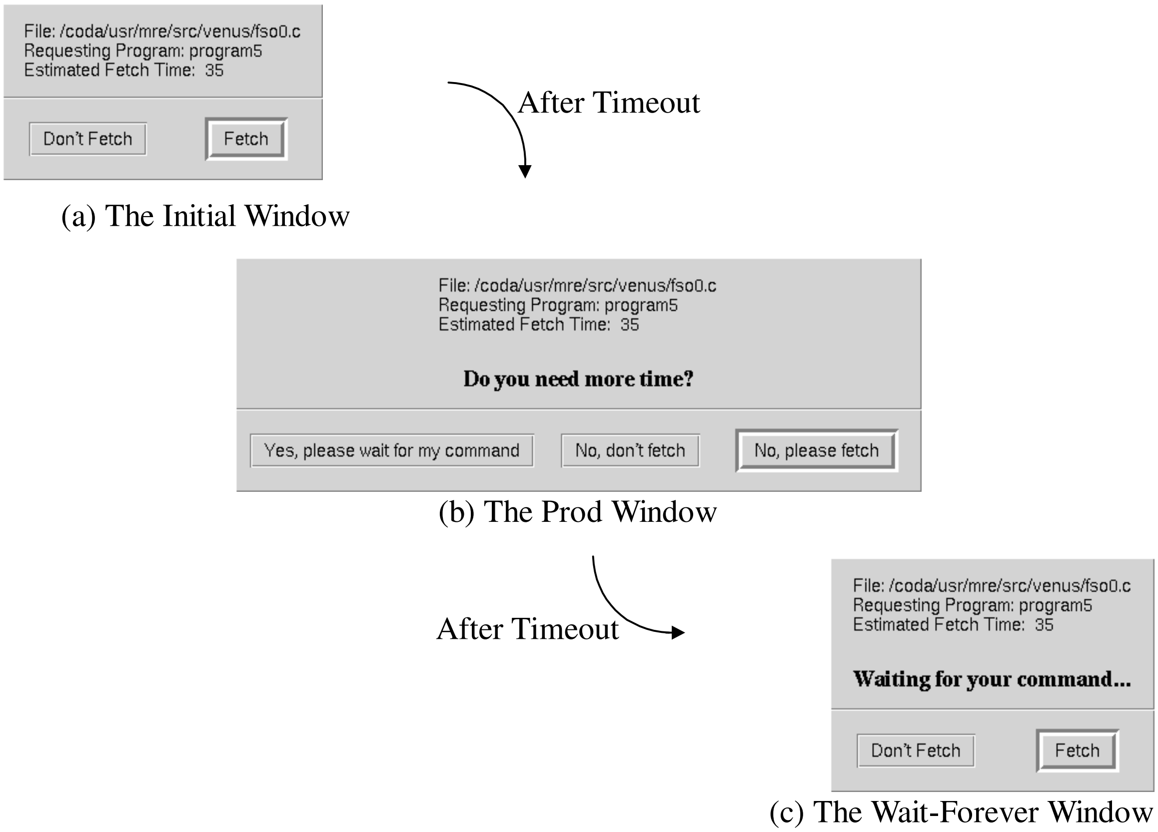


Figure 4.8: Weak-miss Interaction

This figure shows the progression of a weakly connected cache miss query. View (a) is triggered when the user requests (perhaps indirectly) a file system object that is not currently cached and whose estimated fetch time exceeds a certain limit (as described in Section 6.3) while the client is operating weakly connected. This view displays the name of the missing object, the name of the requesting program, and the estimated fetch time. If the user has not answered the query after a certain amount of time (15 seconds by default), then view (a) is replace by view (b). This new view displays the same information, but offers the user the opportunity to consider the request without time pressure. If the user has not responded to this second query within another timeout period, the system will destroy this window and immediately begin fetching this object. If the user wants more time to consider her decision, she can click on the Yes, please wait for my command button. View (b) will then be replaced by view (c) and the system will wait indefinitely for her response.

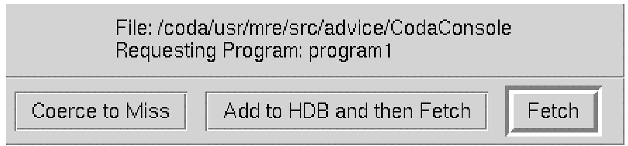


Figure 4.9: Read Disconnected Cache Miss Questionnaire

This figure shows a read disconnected cache miss query. The top portion of the window contains the name of the missing object and of the program requesting it. The buttons at the bottom allow the user to specify what action the system should take. From left to right, the system will coerce the request into a cache miss (and thereby not fetch the file), fetch the file and add the object to the hoard database, or simply fetch the file.

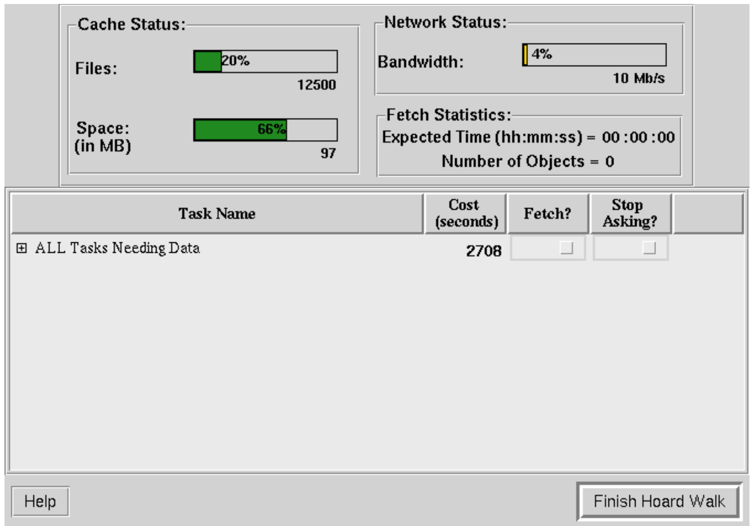
determining when the user returns and is again able to request advice. One solution to this problem is for the interface to pop-up a “Click here when you return” window on the screen. This extension allows the system to handle both attended and unattended operation, while minimizing the time wasted waiting for the user to respond to queries.

4.6.1.3 Read Disconnected Cache Miss Query

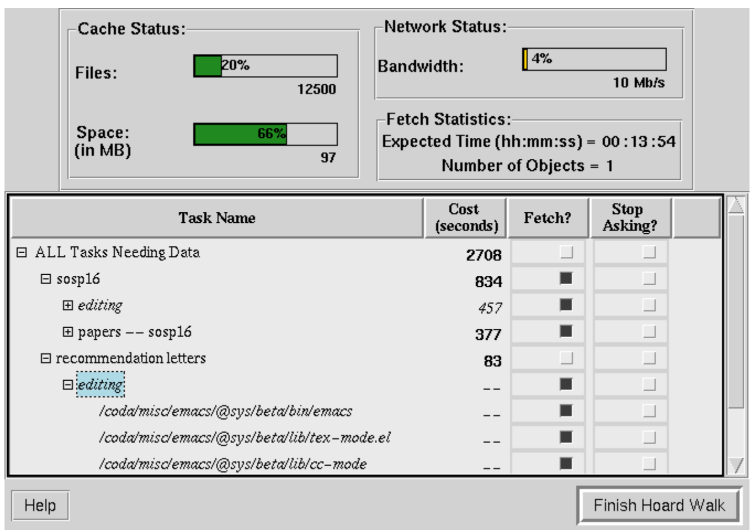
The query that is shown in Figure 4.9 results from a cache miss that occurs while Venus is operating read disconnected (see Section 4.2.4). Read disconnected operation occurs only at the request of the user and helps the user test her hoard database in preparation for a future disconnected session. The purpose of this request is to make cache misses, which are normally transparent to the user, observable. By making cache misses translucent to the user, she is better able to notice omissions in her hoard database. As is true of all the cache miss queries, this request also presents the user with the names of the missing object and the requesting program. In this case, however, the user is presented with three options. She can advise Venus to fetch the object, to fetch the object and add it to the hoard database, or to coerce the request to a miss. The latter option allows the user to determine the effect of not having the object resident in the cache during a disconnected session and allows her to make more informed hoarding decisions.

4.6.1.4 Hoard Walk Advice

The hoard walk advice request occurs during weakly connected operation between the status walk and the data walk when the amount of data to be fetched is expected to take a substantial amount of time. The purpose of this request is to allow the user to control the amount of data fetched over the weak network. After performing the status walk, Venus knows which objects need to be fetched and their sizes. From this information, it can determine the expected fetch times and request advice from the user appropriately. As seen in Figure 4.10, the user is presented with cache status information, current network conditions, total expected fetch time and a hierarchical listing of tasks that require data to



(a) Initial View



(b) Expanded View

Figure 4.10: Hoard Walk Advice

This figure shows a hoard walk advice request. View (a) is displayed initially. The upper left corner shows the status of the cache: the percentage of cache container files and cache blocks dedicated to hoarded objects. The upper right corner shows the current average bandwidth to the servers. The expected time to complete the hoard walk is shown below the network status. The main area of the window shows a list of the tasks that need data to be fetched. Initially, only the "All Tasks Needing Data" task is shown. If the user expands this task as well as the tasks one level below it (by clicking on the "+" signs or double-clicking their names) and then selects the `sosp16` task, the window shown in view (b) will be displayed. Italicized text names indicate a task that is contained in more than one task definition. For each element of this hierarchical list, the expected cost (currently shown only in units of time) is listed to the right of the task name. Further to the right are two checkboxes. The left checkbox allows the user to select the item to be fetched. The one to the right allows the user to instruct the system to not fetch the item during the current hoard walk and, furthermore, to not ask about this item in future hoard walks during this weakly connected session.

be fetched.[12] The task listing is an expandable tree. At the top-most level, a pseudo-task represents all tasks needing data. Expanding this pseudo-task reveals hoarded tasks needing data. The bottom-most level shows individual files and directories that need to be fetched. For any task or object, the user can choose one of three options:

Option:	**Action to choose:**
Do not fetch during this hoard walk	No action required
Fetch during this hoard walk	Click on Fetch? checkbox
Do not fetch until strongly connected	Click on Stop Asking? checkbox

When the user requests that a task or object be fetched, the estimated cost of fetching that task or object is added to the *Fetch Statistics* section of the window. As Figure 4.10(b) shows, *editing* is a subtask of both *sosp16* and *recommendation letters*. Tasks and objects duplicated in other parts of the tree are shown in italics. The cost of fetching these objects is charged only to the first task to request they be fetched as shown by the *editing* subtask in Figure 4.10(b); other tasks obtain such data "for free", as indicated by the "--" in the figure. Once the user has specified those tasks or objects that should be fetched, she can request that the hoard walk complete by clicking on the Finish Hoard Walk button in the lower right corner of the window.

4.6.2 Offering Advice to the User

Venus offers two types of advice to the user. Both relate to task definitions. The first type identifies files that the user may have forgotten to include in a definition. These are files that the user should, perhaps, be hoarding, but is not currently. The second type identifies files that the user may have included unintentionally in a definition or that the user no longer needs. These are files that the user is hoarding, but probably should not be. In both cases, files are identified by heuristics. In this section, I will present the windows used to offer the advice to the user. I will postpone discussing the specific heuristics until Section 6.2.1.

Unlike requests for advice, which are triggered by specific events, offers of advice are presented only when the user specifically requests the information.[13] After the user clicks on the Request Hoard Advice button on the *Advice Information* window, the interface examines usage statistics maintained by both Venus and the advice monitor. The results of this analysis appear in the Advice Offered section of the *Advice Information* window in

[12] The original version of this request, as presented in [34], listed all files and directories needing to be fetched. This design violates the third principle, "speak the user's language", discussed in Chapter 3. By present this information in a task-based hierarchy, not only can the user control how much detail they examine but they also have an idea of whether or not a file is important to their work.

[13] The interface could trigger these offers of advice automatically. Two obvious triggers are after the user requests a hoard walk and after the user reconnects following a disconnected session. I chose not to trigger this advice automatically, however, to minimize the frequency with which we interrupt the user with advice.

the form of two entries: Consider Adding and Consider Removing. The windows associated with these two entries appear in Figure 4.11.

4.6.2.1 Consider Adding

The purpose of this window is to alert the user to file system objects that she might want to hoard. This information is presented in tabular form as seen in Figure 4.11(a). The table shows the pathname of the object as well as an indication of why the object should, perhaps, be hoarded. In addition, each object has an associated check box that allows the user to indicate that the object should not be hoarded (and thus, prevent the system from identifying this object in the future).

4.6.2.2 Consider Removing

The purpose of this window is to alert the user to file system objects that are currently hoarded, but that are not being used. This information is presented in tabular form as seen in Figure 4.11(b). As before, the table shows the pathname of the object, an indication as to why the object should not be hoarded, and a check box that prevents future identification of this object.

4.7 Hoard Walk

The sixth indicator light, labeled *Hoard Walk*, alerts the user to the status of the hoard daemon. When this indicator is shown in green, the hoard daemon is inactive. If the user double-clicks on the indicator light, the window shown in Figure 4.12(a) will appear. When the hoard daemon begins walking the hoard database, the indicator light turns yellow. Double-clicking the indicator during a hoard walk displays the progress meter shown in Figure 4.12(b).

During the course of a hoard walk, a large amount of data may need to be fetched. When Venus is operating weakly connected, a hoard walk can take a substantial amount of time, monopolizing the network for the duration of the walk. To allow the user to control the amount of data actually fetched during any particular hoard walk, Venus asks the user for advice after the hoard daemon completes the status walk and before it begins the data walk. When Venus requests this advice, both the *Hoard Walk* and *Advice* indicator lights turn red. The advice window, shown in Figure 4.10(a), is displayed when the user double-clicks on the *Hoard Walk* indicator. (It is also available through the *Advice* indicator window shown in Figure 4.6.) For more information regarding the specifics of the hoard walk advice request, see Section 4.6.1.4.

Hoard walks occur periodically during the course of normal operation. Currently, the period is approximately 10 minutes. However, at times, users prefer that the hoard daemon walk the database only on demand. For this reason, the *Hoard Walk Information* window allows the user to turn periodic hoard walks off (see Figure 4.12a).

Pathname	Infrequent but Consistent	Accessed (lucky)	Missed (unlucky)	Ignore?
/coda/usr		X		☐
/coda/usr/mre		X		☐
/coda/usr/satya		X		☐
/coda/usr/mre/thesis/dissertation/user_interface.tex			X	☐

Help

(a) Consider Adding Advice

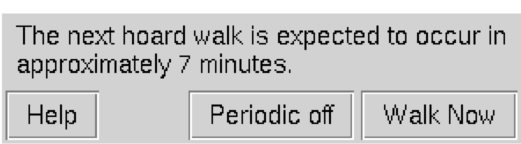
Pathname	Not Accessed Recently	Not Frequently Accessed	Ignore?
/coda/usr/satya/papers/s16/TABS/tsop.aux		X	☐
/coda/usr/satya/papers/s16/TABS/upcall.aux		X	☐
/coda/usr/satya/papers/s16/TABS/tsop.tex		X	☐
/coda/usr/satya/papers/s16/TABS/video-score.tex		X	☐
/coda/usr/satya/papers/s16/TABS/video-drops.aux		X	☐
/coda/usr/satya/papers/s16/TABS/video-score.aux		X	☐
/coda/usr/satya/papers/s16/TABS/video-drops.tex		X	☐
/coda/usr/mre/hoarding/yacc.hdb	X	X	☐
/coda/usr/mre/hoarding/701c.hdb		X	☐

Help

(b) Consider Removing Advice

Figure 4.11: Advice Regarding Hoard Content

This figure shows the windows used to offer advice to the user regarding the content of the hoard database. View (a) suggests files and directories that the user might want to consider adding to one or more task definitions. View (b) suggests files and directories that the user might want to consider removing from one or more task definitions. In both cases, the user is presented with a table of information. The table shows the pathname of the object as well as an indication as to why the given file or directory might need to be added or removed. In addition, each object has an associated check box that allows the user to tell the system to ignore this object in the future.



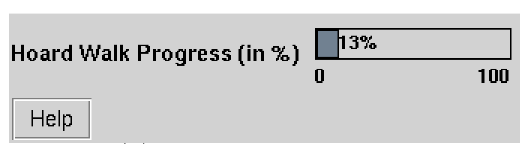


Figure 4.12: Hoard Walk Indicator Windows

This figure shows two views of the *Hoard Walk Information* window. If the hoard daemon is inactive, the window shown in (a) is displayed when the user double-clicks on the *Hoard Walk* indicator light. This window informs the user when the next walk is expected to begin, and allows the user to begin a hoard walk. If the hoard daemon is active, the window shown in (b) is displayed when the user double-clicks on the indicator light. This window gives the user an indication as to the progress of the hoard daemon.

4.8 Reintegration

The seventh indicator light, labeled *Reintegration*, alerts the user to volumes (called *subtrees* in the interface) that need to be reintegrated with the Coda servers. The indicator light turns yellow when a reintegration is currently in progress. The indicator light turns red when a volume needs to be reintegrated but cannot be because the user is not currently authenticated. For example, if the `u.mre` volume (mounted at `/coda/usr/mre`) needs to be reintegrated but user `mre` does not have valid tokens, the reintegration indicator light would turn red. If the user then double-clicked on the indicator light, the window shown in Figure 4.13 would appear. Once all pending reintegrations complete, the indicator light returns to green.

Exposing the state of reintegration to the user is important because it addresses two usability problems observed in the original Coda system. The first of these occurred when users returned from a period of disconnection and failed to authenticate. Without tokens, the client could not propagate the updates to the servers. By turning the "Reintegration" indicator red in this situation, the user is made aware that a reintegration is pending. The second problem occurred when a user who is operating weakly connected makes updates and expects to see those updates reflected on the servers before they have actually been propagated. By turning the indicator yellow, the user can see that a reintegration is in progress. In this case, the indicator serves to influence the user's expectations about the system's behavior and state.

An interesting extension to this indicator light window is to allow the user to have some control over how much of a volume is reintegrated at one time. Such an extension is presented in Section 9.2.3.2, including a paper design of the graphical interface offering this control.

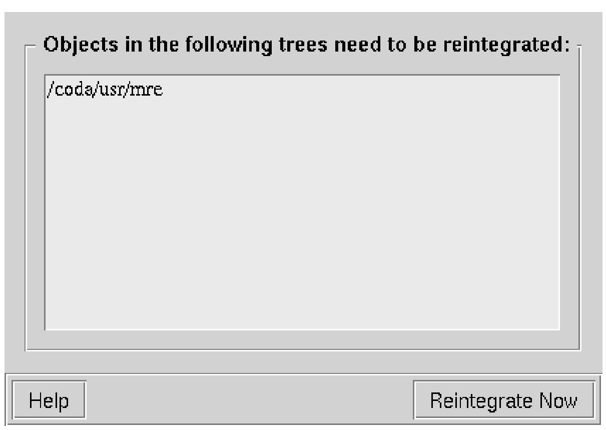


Figure 4.13: Reintegration Information
This figure shows the *Reintegration Information* window. It is displayed when the user double-clicks on the *Reintegration* indicator light. When the indicator light is red, this window shows which subtrees need to be reintegrated. Although Venus performs reintegration at the granularity of volumes, the interface presents the user with the pathname to the mount point of the volume.

4.9 Repair

Coda uses an optimistic replica control strategy. This strategy allows any client to modify data on any accessible server. One consequence of this design is that replicas may come in conflict with one another. The situations that lead to conflict are infrequent, though not impossible, in the environments for which Coda is most appropriate. When a file or directory becomes inconsistent, the system attempts to resolve the conflict automatically. Should automatic resolution fail, the user must manually repair the problem.

The eighth indicator light, labeled *Repair*, alerts the user to file system objects that are currently in conflict. The indicator light turns red when an object needs to be repaired manually and its supporting window lists those objects needing such repair. For example, if the directory `/coda/usr/mre/thesis/dissertation` were to become in conflict, the *Repair* indicator light would turn red and the window shown in Figure 4.14 would appear when the user double-clicked on the indicator light. Once all known objects needing repair are fixed, the indicator light returns to green. There is no notion of yellow for this indicator.

A simple, but useful, extension to the interface would present summary information about how the object came to be in conflict. In particular, the user might appreciate knowing who updated the files and from where those updates were made. Such information would help the user understand why the object is in conflict and would also help them to resolve the conflict. This data, however, is not readily available in Coda.

A further extension to this indicator light window is to offer a graphical interface to the repair program. When the window shown in Figure 4.14 is visible, the user could double-click on any object needing to be repaired. This action would open a repair window that would allow the user to repair the object. This extension is discussed in Section 9.2.3.1.

Figure 4.14: Repair Information

This figure shows the *Repair Information* window. It is displayed when the user double-clicks on the *Repair* indicator light. When the indicator light is red, this window shows which objects are in conflict and need to be repaired.

4.10 Task

As discussed in Sections 3.1.3.1 and 3.4.2.1, I must improve the model through which users identify what objects should be available during a disconnected or weakly connected session. The realization of that model appears in the final indicator light, which is labeled *Task*. This indicator allows the user to define and hoard tasks, and also alerts the user to the availability of those tasks. A task is simply the file and directories a user needs to complete a given project (e.g., writing a paper or compiling a system). It is a hierarchical structure consisting of user data, programs, and other tasks.

When the indicator is shown in green, the user knows that all hoarded tasks are currently available. When the indicator is shown in red, the user knows that at least one hoarded task is partially or completely unavailable. The indicator is never shown in yellow.

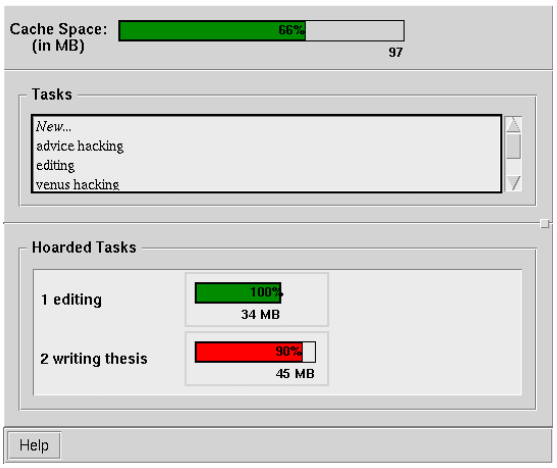
Double-clicking on the indicator light will cause a window similar to that shown in Figure 4.15(a) to appear. This window allows users to hoard (and unhoard) tasks. The top of this window shows the percentage of cache space dedicated to hoarded tasks. The middle section shows a list of all defined tasks as well as the pseudo-task "New…" that allows the user to define new tasks. The bottom portion of this window shows those tasks that are currently hoarded in priority order, where the task identified with a "1" is the highest priority.

Double-clicking on the name of a task in either of the lists of Figure 4.15(a) will display a window similar to that shown in Figure 4.15(b), allowing the user to view, modify, or create a task definition. A task has three main components: data, programs, and subtasks. For each of these components, the task definition shows two lists. The *predefined* list contains a list of all currently defined data sets, programs or tasks, respectively. The *contains* list shows those data sets, programs or tasks that are included in the current definition. Clicking on the name of an element in a predefined list includes it in the current task definition.

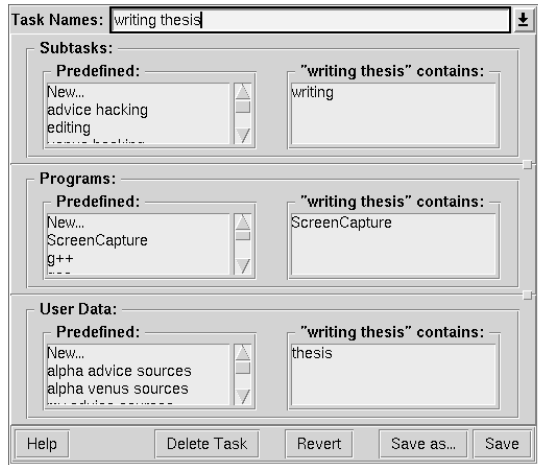
Double-clicking on an element shown in the predefined or contains lists causes a window containing the appropriate definition to appear, allowing the user to view a definition or create a new one. In particular, double-clicking on the name of a data definition will cause a window similar to that shown in Figure 4.15(c) to appear. Similarly, double-clicking on the name of a program definition will cause a window similar to that shown in Figure 4.15(d) to appear.

The data definition window allows users to specify which of their files and directories should be included within a task definition. If the pathname is a directory, then the user may specify whether or not the children and/or descendants of this directory should also be hoarded. This window is essentially a graphical interface to the hoard program, as described in Section 2.3.2. Unlike previous versions of Coda, the user is required to have detailed knowledge only of their own areas of the file system.

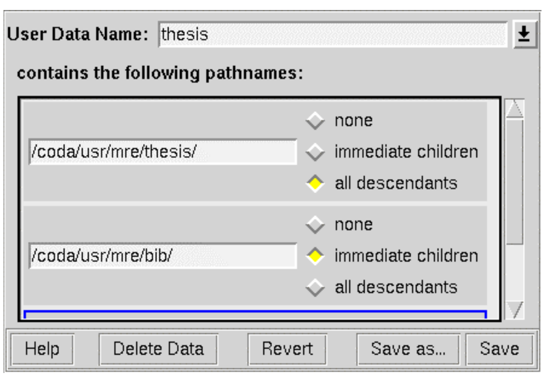
The program definition window allows users to specify programs that should be included within a task definition. Once included, the system automatically tracks file references



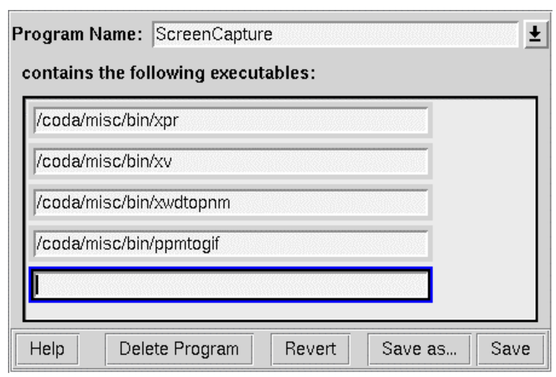
(a) Task Information Window



(b) Task Definition Window



(c) Data Definition Window



(d) Program Definition Window

Figure 4.15: The Windows Associated with the Task Indicator

This figure shows the four primary windows associated with the *Task* indicator light. View (a) shows the window displayed after clicking on the indicator. The top section of this window shows the status of the cache; the middle section shows a list of all defined tasks; and the bottom section shows those tasks that have been hoarded, their priorities, and their availability. View (b) shows the definition for the “writing thesis” task. This definition contains the “writing” task, the programs for “ScreenCapture”, and the “thesis” user data set. View (c) shows the definition for the “thesis” user data. This definition includes the `/coda/usr/mre/thesis` directory and all its descendants as well as the `/coda/usr/mre/bib` directory and its immediate children. View (d) shows the definition for the “ScreenCapture” programs, including `xpr`, `xv`, `xwdtopnm`, and `ppmtogif`.

made by the program for automatic hoarding. The details of how automatic program hoarding works will be discussed in Section 6.1. With automatic program hoarding, users are no longer required to specify the files and directories required internally by these programs (e.g., library, font, and style files).

The distinction between user data and program data is crucial for reducing the burden of hoarding. Although it is not unreasonable to assume that users have detailed knowledge of their own areas of the file system, they must not be required to learn about the internal workings of application programs to make effective use of Coda. By distinguishing between user and program data, the interface allows the program profiles to be shared between tasks (e.g., not specialized to a single project). Of course, if the entire hoarding process were automated in such a way that the tasks were meaningful to users, even this burden could be eliminated. This interface does not in any way preclude such automation; in fact, this is an area identified for future research and is discussed in Section 9.2.2.4.