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.

![Figure 4.1: Indicator Lights](http://www.cs.cmu.edu/Reports)

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.
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. 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.
4.4 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. 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.
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.
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 (of 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](image)

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.
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. 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:

<table>
<thead>
<tr>
<th>Option</th>
<th>Action to choose:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do not fetch during this hoard walk</td>
<td>No action required</td>
</tr>
<tr>
<td>Fetch during this hoard walk</td>
<td>Click on Fetch? checkbox</td>
</tr>
<tr>
<td>Do not fetch until strongly connected</td>
<td>Click on Stop Asking? checkbox</td>
</tr>
</tbody>
</table>

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. 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.
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
Interacting with the User

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.
5 Implementation

In the previous chapter, I described the graphical user interface that makes caching translucent to the user. In this chapter, I focus on the implementation of that interface. The first section presents an overview of the system’s architecture. I continue this chapter by describing a number of implementation details. These details include descriptions of some finite state machines that control the user interface as well as an overview of the modifications required within Venus. I conclude the chapter with an overview of the Translucent Caching API. This API allows Venus to notify the CodaConsole of important events and allows the CodaConsole to make the user’s wishes known to Venus.

5.1 Architecture of System

Logically, the functionality of the CodaConsole belongs within Venus (see Chapter 2) because Venus alone has knowledge of the various events needed to drive the system. However, other considerations led me to make this functionality external to Venus. The benefits of translucent caching extend beyond Coda. Moving this functionality external to Venus allows the CodaConsole to be used with any highly available file system implementing its API. A second consideration was purely practical. Venus is a large and complex body of code. Further, at the time of this research, it was under development by a number of individuals. Making my interface as independent of Venus as possible isolated me from these other development efforts. For these reasons, the CodaConsole is located outside the scope of Venus and communicates with Venus via a well-defined API.

Placing the CodaConsole external to Venus imposes additional communication overhead by adding a local RPC2 call [45]. Because daemons invoke many of these calls, the user does not directly observe this additional overhead. Those calls that are invoked in the process of servicing a user request are generally infrequent or long running; thus, the additional overhead is inconsequential. A few calls would impose too great a burden and are thus batched.

The architecture, shown in Figure 5.1, is implemented in three pieces. I refer to the first two pieces, the user interface and the Advice Monitor, jointly as the CodaConsole. They implement the graphical user interface described in the previous chapter. The user interface interacts with the user. The Advice Monitor acts as a liaison between this user interface and Venus. The third piece consists of a set of hooks within Venus that inform the CodaConsole of various events.
This diagram shows the architecture of the system. The CodaConsole consists of two parts: the user interface and the Advice Monitor. The user interface implements the graphical interface described in the previous chapter. The Advice Monitor acts as a liaison between Venus and this interface. Venus notifies the Advice Monitor of various events via the RPC interface implementing the Translucent Caching API.

These three pieces communicate using two different mechanisms. The user interface communicates with the Advice Monitor via a pair of unidirectional pipes, as shown in the figure. The Advice Monitor communicates with Venus via a pair of RPC2 connections, also shown in the figure.

### 5.2 Details of Implementation

This section describes the implementation of each of the three components described above. I begin by describing the implementation of the user interface, focusing on the finite state machines that drive the indicator lights. I then present a brief description of the Advice Monitor. I conclude with a summary of the modifications necessary to implement the Translucent Caching API within Venus.
5.2.1 User Interface

Chapter 1 described the design of the user interface component of the CodaConsole. The user interface, which is implemented in Tcl/Tk [39, 56] using the Tix widget library [29], assumes the user is running a windowing system. It is entirely event driven. As events arrive, the interface notifies the user via the indicator lights. As the user requests information, it displays the appropriate information windows. If it needs data from Venus, it contacts the Advice Monitor to request that data.

The interface presented to the user contains a set of indicator lights. These indicator lights are implemented as small, event-driven, finite state machines. The arrival of an event potentially causes a transition from one state to another. These state changes may then cause the appearance of the indicator lights to change. I will now describe three of these state machines. The remaining ones are similar in spirit, though the details differ.

5.2.1.1 Tokens Indicator

The finite state machine for the Tokens indicator has three states: Valid, Invalid, and Expired&Pending. The user is required to have tokens before the advice monitor begins executing so the finite state machine will start in the Valid state. When the interface receives notification that the user’s tokens have expired, the machine transitions into the Invalid state. If the interface then receives a notification that an activity, a reintegration for example, is pending tokens, the machine transitions into the Expired&Pending state. The machine transitions to the Valid state from either of the other two states upon receiving notification that the user has obtained valid tokens. These states and the transitions between them are shown in Figure 5.2.

![Figure 5.2: State Machine of the Tokens Indicator](image)

---

14 Currently, the system works under X windows, though it does not require that windowing system.
5.2.1.2 Hoard Walk Indicator

The finite state machine of Figure 5.3 drives the Hoard Walk indicator and contains four states: Inactive, Active, Stalled, and Suspend. The machine initially begins in the Inactive state. If the interface is then notified that a hoard walk has begun, the machine transitions to the Active state. Should Venus request advice, the machine transitions to the Stalled state until the user provides that advice and causes the machine to transition back to the Active state. When Venus announces the completion of the hoard walk, the machine transitions back to the Inactive state. Should the user request that Venus refrain from performing periodic hoard walks, the machine will transition to the Suspend state until periodic hoard walks are once again enabled.

5.2.1.3 Task Indicator

A smaller but slightly more complex pushdown automaton drives the Task indicator. This machine, which is shown in Figure 5.4, contains just two states (All Tasks Available and At Least One Task Unavailable), a counter, and a list of available tasks. At the end of each hoard walk, Venus announces the availability of each hoarded task owned by the user. This availability information is presented to the user in the Task Information window.

Suppose that Venus just completed a hoard walk and announced that all hoarded tasks were available. Further suppose that a hoarded file is invalidated (because, for instance, it was updated remotely). Venus would then announce that an object has become unavailable. In this message, it would also specify the task and the size of the object. The interface would now transition to the At Least One Task Unavailable state, set the counter of unavailable tasks to 1, remove this task from the list of available tasks, update the information about this task, and, finally, notify the user of this event. Now suppose that Venus must replace a hoarded file (for instance, due to space limitations). When this event occurs, Venus announces that the object has become unavailable as before. The interface, however, must be more careful in responding to this event. It must first determine whether or not this task was already unavailable. If the task was previously unavailable, then the system simply adjusts the percentage of the task available based upon the new information. If the task was previously available, then the system increments the counter of unavailable tasks, removes this task from the list of available tasks, and adjusts the percentage of the task available. Because the interface is already in the At Least One Task Unavailable state, it does not take a transition.

5.2.2 Advice Monitor

The primary purpose of the Advice Monitor is to act as a liaison between the user interface and Venus. As events arrive from Venus, the Advice Monitor translates the request into an appropriate procedure call within the interface, and then ships that call to
Figure 5.3: State Machine of the Hoard Walk Indicator

Figure 5.4: State Machine of the Task Indicator
the interface. As requests arrive from the interface, the Advice Monitor translates the request into an appropriate RPC2 call.

The Advice Monitor also has a number of secondary responsibilities. These include moving data files (the results of small questionnaires) into the Coda file system, processing certain usage statistics provided by Venus for the purpose of offering advice (as discussed in Sections 4.6.2 and 6.2), and controlling the Application Specific Resolvers, as described in [28].

The Advice Monitor uses a total of eight light-weight processes [45], or threads, to fulfill its obligations. Each thread and its primary responsibilities are described below:

Main: Performs initialization, loops waiting for incoming RPC requests and incoming messages from the user interface, and also dispatches other threads

CodaConsoleHandler: Handles messages arriving from the user interface, making the appropriate RPC to Venus (if required)

WorkerHandler: Handles RPCs arriving from Venus

ProgramLogHandler: Analyzes incoming program logs

ReplacementLogHandler: Analyzes incoming replacement logs, adding entries to the database

DataHandler: Moves data from temporary locations into permanent storage in Coda file system

EndASREventHandler: Watches for completion of Application-Specific Resolvers (ASRs) and returns the results to Venus

ShutdownHandler: Handles shutdown requests, closing connections and exiting the user interface

Two of these threads warrant further attention and will be discussed in the following chapter. These two threads, the Program Log Handler and the Replacement Log Handler, help to reduce the burden hoarding places on users.

5.2.3 Venus Modifications

Supporting the CodaConsole required modifications to Venus in two areas. First, I added the advice module, which implements the Venus side of the Translucent Caching API. The advice module exposes certain Venus data to the user and provides wrappers around the calls of the API. Second, I added hooks throughout Venus that invoke the wrappers within the advice module. These hooks expose certain events to the user by notifying the CodaConsole. This section describes both modifications.
Figure 7.33: Finding Originations
This figure shows where evidence for each finding of Appendix F originated. The four top-level nodes correspond to the tutorial (T), exercises (E), questionnaire (Q), and transcripts of verbal protocol (V). Each leaf in the tree represents a single finding. The color of the leaves represents the severity rating of the given finding. Red indicates a Level 1 severity rating; yellow represents a Level 2 finding; green represents a Level 3 finding; and, blue represents a Level 4 finding. The size of the leaf represents the scope of the finding. Wide leaves correspond to global findings; narrow leaves correspond to local findings.
7.7.4 Optimal Number of Participants

Given the amount of effort needed to observe and analyze the data from each participant in a usability test, it comes as no surprise that the HCI community would like to know the optimal number of participants needed. I analyzed the evidence from each of the findings to determine which users contributed any piece of evidence, no matter how minor. This analysis is shown graphically in Figure 7.34. Each of the thirteen users is represented by a column in this matrix and each of the findings is represented by a row. Each colored box in this figure indicates that the given user contributed evidence for the given finding. More informative users appear toward the right. More frequently reported problems appear toward the top. The figure shows that the most informative user contributed to more than 50% of the findings while the least informative user contributed to fewer than 10% of the findings. The color of the box indicates the severity of the finding. This figure shows that many users found evidence for most, but not all, severe problems. In fact, six of the most severe problems were reported by just two users apiece. It also shows that most of the Level 4 problems were reported by just a few users.

Assuming that any piece of evidence would be sufficient to identify a problem, I graphed the percentage of findings uncovered as my test progressed (i.e., with my users in the order they appeared in the test). I examined the findings at each individual level, as well as the most critical problems (those with severity levels 1 or 2) and the findings as a whole. These data are shown in Figure 7.35(a-f).

An interesting characteristic of these graphs is that all but one of them is best approximated by a log-based function. The surprise is that one of them, the severity level 4 findings, is best approximated by a linear function. Evidently each additional user saw different enhancements. I hypothesize that this curve might also be logarithmic in the limit; in fact, you can see what might be a flattening trend near the end.

With just three users, I would have uncovered more than 50% of the problems at severity levels 1 and 3, but only 30% of the problems at the other levels. To uncover 80% of the problems, I would have had to run approximately six users. It is clear that beyond about six users I quickly reached the point of diminishing returns.

My next goal was to determine what, if any, effect the ordering of participants had on these results. To make this determination, I calculated the number of findings uncovered by every subset of my users. I then calculated the average number of findings reported by subsets of a given size. For instance, a set of thirteen users contains 78 subsets of size two. For each of these subsets, I calculated the number of findings they uncovered. I then averaged these values over all 78 subsets. By performing this same calculation for the other subset sizes, I could determine the average number of findings uncovered with increasing numbers of participants. On average, I find that six users were required to uncover 80% or more of the findings, but that only three users were required to uncover 50% or more of the findings.
Figure 7.34: Frequency of Reporting
This matrix indicates which users contributed evidence to which findings. Each box indicates that the given user contributed evidence to the given finding. The color of the box indicates the severity of the finding: red represents level 1 problems, yellow represents level 2, green represents level 3, and blue represents level 4. The matrix has been sorted so that more productive users appear in the rightmost columns and more frequently reported problems appear towards the top. The order of users from left to right is: P2, S1, N4, P3, P5, C1, N1, E2, P4, T1, N2, E1, E3.
Figure 7.35: Number Users Required to Cover Most Severe Findings
This figure shows the number of participants necessary to cover the usability findings. In Views (a)-(d), I show just the findings with severity levels 1-4 respectively. In View (e), I show the most critical findings, those with a severity level of 1 or 2. In View (f), I show all findings. In each view, I show the closest function that approximates the data. All but one graph is most closely approximated by a log function. View (d), however, is more closely approximated by a linear function.
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]).
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.